# Pythonwhat Documentation

***Release 1.2.0***

**Vincent Vankrunkelsven**

**Apr 19, 2017**

# Contents

CHAPTER 1

# Contents:

## Home

At DataCamp we build tools to learn data science interactively. See e.g. our online R tutorial to learn R Programming and our Python For Data Science tutorial to learn Python.

### pythonwhat?

A major part of DataCamp's interactive learning is centered around automated and meaningful feedback. When a student submits an incorrect answer, the system tells the student what he or she is doing wrong. This happens through so-called submission correctness tests, or SCTs. An SCT is a test script that compares the different steps in a student's submission to the ideal solution, and generates meaningful feedback along the way.

`pythonwhat` is a Python package that can help you write these SCTs for interactive Python exercises on DataCamp. It allows you to easily compare parts in the student's submission with the solution code. `pythonwhat` provides a bunch of functions to test object definitions, function calls, function definitions, for loops, while loops, and many more. `pythonwhat` automatically generates meaningful feedback that's specific to the student's mistake; you can also choose to override this feedback with custom messages.

Writing SCTs, for which `pythonwhat` is built, is only one part of creating DataCamp exercises. For general documentation on creating Python courses on DataCamp, visit the Teach Documentation. To write SCTs for R exercises on DataCamp, have a look at testwhat.

### How does it work?

When a student starts an exercise on DataCamp, a Python session is started and the `pre_exercise_code` (PEC) is run. This code, that the author specifies, initializes the Python workspace with data, loads relevant packages etc, such that students can start coding the essence of the topics treated. Next, a separate solution process is created, in which the same PEC and actual solution code, also coded by the author, is executed.

When a student submits an answer, his or her submission is executed and the output is shown in the IPython Shell. Then, the correctness of the submission is checked by executing the Submission Correctness Test, or SCT. Basically, your SCT is a Python script with calls to `pythonwhat` test functions. `pythonwhat` features a variety of functions to test a user's submission in different ways; examples are `test_object()`, `test_function()` and `test_output_contains()`. To do this properly, `pythonwhat` uses several resources:

- The student submission as text, to e.g. figure out which functions have been called.
- The solution code as text, to e.g. figure out whether the student called a particular function in the same way as it is called in the solution.
- The student process, where the student code is executed, to e.g. figure out whether a certain object was created.
- The solution process, where the solution code is executed, to e.g. figure out whether an object that the student created corresponds to the object that was created by the solution.
- The output that's generated when executing the student code, to e.g. figure out if the student printed out something.

If, during execution of the SCT, a test function notices a mistake, an appropriate feedback will be generated and presented to the student. It is always possible to override these feedback messages with your own messages. Defining custom feedback will make your SCTs longer and they may be error prone (typos, etc.), but they typically give the exercise a more natural and personalized feel.

If all test functions pass, a success message is presented to the student. `pythonwhat` has some messages in store from which it can choose randomly, but you can override this with the `success_msg()` function.

## Overview

To get started, make sure to check out the *Quickstart Guide*.

To robustly test the equality of objects, and results of evaluations, it has to fetch the information from the respective processes, i.e. the student and solution processes. By default, this is done through a process of 'dilling' and 'undilling', but it's also possible to define your own converters to customize the way objects and results are compared. For more background on this, check out *Managing Processes*. For some more background on the principle of 'sub-SCTs', i.e. sets of tests to be called on a particular part or a particular state of a student's submission, have a look at the *Part Checks article*.

The remainder of the wiki goes over every test function that `pythonwhat` features, explaining all arguments and covering different use cases. They will give you an idea of how, why and when to use them.

For more full examples of SCTs for Python exercises on DataCamp, check out the source files of the pythonwhat tutorial course. In the chapter files there, you can can see the SCTs that have been written for several exercises. Another useful course is introduction to Python.

After reading through this documentation, we hope writing SCTs for Python exercises on DataCamp becomes a painless experience. If this is not the case and you think improvements to `pythonwhat` and this documentation are possible, please let us know!

## Quickstart Guide

### Course Setup

This guide will cover the basics of creating submission correctness tests (SCTs) for DataCamp exercise. SCTs deal with the running and testing code submissions, in order to give useful feedback. For help on the entire exercise creation process, check out https://www.datacamp.com/teach/documentation. If this is your first time creating a course, see their Getting Started screencast and Code Exercises article.

## Your First Exercise

As a basic example, suppose we have an exercise that requires the student to print a variable named `x`. This exercise could look something like this:

~~~
    *** =pre_exercise_code
    ```{python}
    x = 5
    ```

    *** =sample_code
    ```{python}
    # Print x

    ```

    *** =solution
    ```{python}
    # Print x
    print(x)
    ```

    *** =sct
    ```{python}
    Ex().check_object('x').has_equal_value()
    Ex().test_output_contains('5')
    success_msg('Great job!')
    ```
~~~

The SCT uses three `pythonwhat` chains to test the correctness of the student's submission.

1. `check_object` is used to test whether `x` was defined in the submission. In addition, the `has_equal_value()` statement tests whether the value of `x` is equal between the submission and solution.
2. `test_output_contains()` tests whether the student printed out `x` correctly. The function looks at the output the student generated with his or her submission, and then checks whether the string '5' is found in this output.
3. `success_msg()` is used to give positive feedback when all `pythonwhat` tests passed. If you do not use `success_msg()`, `pythonwhat` will generate a kind message itself :).

In all the test statements above, feedback messages will be automatically generated when something goes wrong. However, it is possible to manually set these feedback messages. For example, in the code below,

```
Ex().check_object(missing_msg="`x` is undefined!") \
    .has_equal_value(incorrect_msg="wrong value for `x`")
```

the automatic messages for when `x` is undefined or incorrect are replaced with manual feedback. Now, if students submit `x = 4` instead of `x = 5`, they will see the message, "wrong value for `x`". Finally, notice that you can use Markdown syntax inside the strings here.

The same holds for `has_equal_value()`: you can use the `incorrect_msg` argument to specify a custom message. For more information on all the different arguments you can set in the different `pythonwhat` functions, have a look at the articles in this wiki, describing them in detail.

## Next Steps

Test functions in pythonwhat are broken into 4 groups:

- *Simple tests*: look at, e.g., the output produced by an entire code submission.
- *Part checks*: focus on specific pieces of code, like a particular for loop.
- *Expression tests*: combined with part checks, these run pieces of code and evaluate the outcome.
- *Logic tests*: these allow logic like an or statement to be used with SCTs.

## Simple Tests

Simple tests are the most basic tests available in pythonwhat. They usually don't focus on specific pieces of a submission (like part checks), or re-run any code (:doc:'like expression tests </expression_tests.md>). Instead, they simply look at things like imports, printed output, or raw code text. A final, common use is to test the value of a variable in the final environment (that is, after the submission of solution code have been run).

### test_import

**test_import** (*name*, *same_as=True*, *not_imported_msg=None*, *incorrect_as_msg=None*, *state=None*)

Test import.

Test whether an import statement is used the same in the student's environment as in the solution environment.

**Parameters**

- **name** (*str*) – the name of the package that has to be checked.
- **same_as** (*bool*) – if false, the alias of the package doesn't have to be the same. Defaults to True.
- **not_imported_msg** (*str*) – feedback message when the package is not imported.
- **incorrect_as_msg** (*str*) – feedback message if the alias is wrong.

**Example** Student code:

```
import numpy as np
import pandas as pa
```

Solution code:

```
import numpy as np
import pandas as pd
```

SCT:

```
test_import("numpy")  # pass
test_import("pandas") # fail
test_import("pandas", same_as = False) # pass
```

```
    def test_import(name,
                    same_as=True,
                    not_imported_msg=None,
                    incorrect_as_msg=None):
```

With `test_import` you can test whether a student correctly imported a certain package. As an option, you can also specify whether or not the same alias should be used.

Python features many ways to import packages. All of these different methods revolve around the `import`, `from` and `as` keywords. Suppose you want students to import `matplotlib.pyplot` as `plt` (the common way of importing the plotting tools in `matplotlib`. A possible solution of your exercises could be the following:

~~~
    *** =solution
    ```{python}
    # Import plotting tools
    import matplotlib.pyplot as plt
    ~~~
```

Below is a possible SCT for this exercise:

```
    *** =sct
    ```{python}
    test_import("matplotlib.pyplot")
    success_msg("You nailed it!")
    ```
```

Here, `test_import` will parse both the student's submission as well as the solution, and figure out which packages were imported and how. Next, it checks if the `matplotlib.pyplot` package was imported and under which alias. If the student did this and imported it as `plt`, all is good. If, however, the student submitted `import matplotlib` (import entire package instead of module) or `import matplotlib.pyplot as pppplot` (incorrect alias), `test_import()` will fail and generate the appropriate messages.

As usual, you can override these messages with your own:

```
    *** =sct
    ```{python}
    test_import("matplotlib.pyplot"),
                                not_imported_msg = "You can import pyplot by using␣
↪`import matplotlib.pyplot`.",
                                incorrect_as_msg = "You should set the correct alias for␣
↪`matplotlib.pyplot`, import it `as plt`.")
    success_msg("You nailed it!")
    ```
```

With `same_as`, you can control whether or not the alias should be exactly the same. By default `same_as=True`, so the alias (`plt` in the example) should also be used by student. If you set it to `False`:

```
    *** =sct
    ```{python}
    test_import("matplotlib.pyplot", same_as=False)
    success_msg("You nailed it!")
    ```
```

The SCT will also pass if the student uses `import matplotlib.pyplot as pppplot`, a submission that wouldn't be accepted if `same_as=True`.

## test_object

**`test_object`**(*name*, *eq_condition='equal'*, *eq_fun=None*, *do_eval=True*, *undefined_msg=None*, *incorrect_msg=None*, *state=None*)

Test object.

The value of an object in the ending environment is compared in the student's environment and the solution environment.

**Parameters**
```

- **name** (*str*) – the name of the object which value has to be checked.
- **eq_condition** (*str*) – how objects are compared. Currently, only "equal" is supported, meaning that the object in student and solution process should have exactly the same value.
- **do_eval** (*bool*) – if False, the object will only be checked for existence. Defaults to True.
- **undefined_msg** (*str*) – feedback message when the object is not defined
- **incorrect_msg** (*str*) – feedback message if the value of the object in the solution environment doesn't match the one in the student environment.

**Example** Student code:

```
a = 1
b = 5
```

Solution code:

```
a = 1
b = 2
```

SCT:

```
test_object("a") # pass
test_object("b") # fail
```

Note that the student code below would fail both tests:

```
a = 1
b = 2
a = 3 # incorrect final value of a
```

```
test_object(name,
            eq_condition="equal",
            do_eval=True,
            undefined_msg=None,
            incorrect_msg=None)
```

`test_object()` enables you to test whether a student correctly defined an object.

As explained on the docs home, both the student's submission as well as the solution code are executed, in separate processes. `test_object()` looks at these processes and checks if the object specified in `name` is available in the student process. Next, it checks whether the object in the student and solution process correspond. In case of a failure along the way, `test_object()` will generate a meaningful feedback message that you can override.

## Example 1

Suppose we have the following solution:

~~~
*** =solution
```{python}
# Create a variable x, equal to 3 * 5
x = 3 * 15
```
~~~

To test this we simply use:

~~~
*** =sct
```{python}
test_object("x")
success_msg("Great job!")
```
~~~

This SCT will test if the variable `x` is defined, and has the same ending value in the student process as in the solution process. All of the following student submissions would be accepted by `test_object()`:

- `x = 15`
- `x = 12 + 3`
- `x = 3; x += 12`

How the object `x` came about in the student's submission, does not matter: only the end result, the actual content of `x`, matters.

`do_eval=True` by default; if you set it to `False`, only the existence of an object `x` will be checked; its contents will not be compared to the object `x` that's in the solution process.

### Object equality

When comparing more complex objects in Python, chances are they don't use the equality operation you desire. Python objects are compared using the == operator, and objects can overwrite its implementation to fit the object's needs. Internally, `test_object()` uses the == operation to compare objects, this means you could encounter undesirable behaviour. Sometimes == just compares the actual object instances, and objects which are semantically alike wont be according to `test_object()`.

Say, for example, that you have the following solution:

```
*** =solution
from urllib.request import urlretrieve
fn1 = 'https://s3.amazonaws.com/assets.datacamp.com/production/course_998/datasets/
↪Chinook.sqlite'
urlretrieve(fn1, 'Chinook.sqlite')

# Import packages
from sqlalchemy import create_engine
import pandas as pd

# Create engine: engine
engine = create_engine('sqlite:///Chinook.sqlite')

# Open engine connection
con = engine.connect()
```

An SCT for this exercise could be the following:

```
*** =sct
test_object("engine")
test_object("con")
```

Now, if the student enters the exact same code as the solution, the SCT will still fail. How? Well if you try this out: `create_engine('sqlite:///Chinook.sqlite') == create_engine('sqlite:///Chinook.sqlite')` you will notice that it returns `False`. This means the exact same execution doesn't lead to the the exact same object (although they might be semantically equal). We can't use `test_object` like that here. There are several ways to solve this:

#### Workaround

```
*** =sct
test_object("engine", do_eval=False)
test_function("create_engine")
test_object("con", do_eval=False)
test_function("engine.connect")
```

This will check whether the objects `engine` and `con` are declared, without checking for it's value. With `test_function()` we check whether they used the correct functions. This will not test the exact same thing as the first SCT, but it's effective 99% of the time.

#### Equality operations hardcoded in `pythonwhat`

A side note here, complex objects that are used a lot have an custom implementation of equality built for them in `pythonwhat`. These objects can be tested with a regular `test_object(...)`, without having to use `do_eval=False`. At the moment, the more complex classes that can be tested are:

- `numpy.ndarray`
- `pandas.DataFrame`
- `pandas.Series`

Of course primitive classes like `str`, `int`, `list`, `dict`, ... can be tested without any problems too, as well as objects of which the class has a semantically correct implementation of the == operator.

#### Manually define a converter

As explained in the *Processes article*, objects are extracted from their respected processes by 'dilling' and 'undilling' them. However, you can manually set a 'converter' with the `set_converter()` function. This will override the default dilling and undilling behavior, and enables you to make simplified representations of custom objects, testing only exactly what you want to test.

**NOTE**: Behind the scenes, `pythonwhat` has to fetch the value of objects from sub-processes. The required 'dilling' and 'undilling' can cause issues for exotic objects. For more information on this and possible errors that can occur, read the *Processes article*.

### check_object

**`check_object`**(*index*, *missing_msg='FMT:Are you sure you defined the {typestr}, '{index}'?'*, *expand_msg='FMT:Check the variable '{index}'. '*, *state=None*, *typestr='variable'*)

### is_instance

**`is_instance`**(*inst*, *not_instance_msg='FMT:Is it a {inst.__name__}?'*, *name=None*, *state=None*)

### has_key

**`has_key`**(*key*, *key_missing_msg="__JINJA__:There is no {{ 'column' if 'DataFrame' in parent.typestr else 'key' }} inside {{parent.index}}."*, *name=None*, *state=None*)

## has_equal_key

**has_equal_key**(*key*, *incorrect_value_msg='FMT: Have you specified the correct value for "{key}" inside '{parent[sol_part][name]}'?'*, *key_missing_msg="__JINJA__:There is no {{ 'column' if 'DataFrame' in parent.typestr else 'key' }} inside {{parent.index}}."*, *name=None*, *state=None*)

## test_output_contains

**test_output_contains**(*text*, *pattern=True*, *no_output_msg=None*, *state=None*)

Test the output.

Tests if the output contains a (pattern of) text.

**Parameters**

- **text** (*str*) – the text that is searched for
- **pattern** (*bool*) – if True, the text is treated as a pattern. If False, it is treated as plain text. Defaults to False.
- **no_output_msg** (*str*) – feedback message to be displayed if the output is not found.

```
test_output_contains(text,
                     pattern=True,
                     no_output_msg=None)
```

We can test the output of the student contains with `test_output_contains()`. This function will compare the given text with the text in the student's output and see if we have a match. You can use regular expressions or not, that's completely up to you.

Here's an example of an exercise with `test_student_typed()`, suppose the solution looks like this:

~~~
*** =solution
```{python}
# Print the "This is some ... stuff" to the shell
print("This is some weird stuff")
~~~
```

The following SCT tests whether the student outputs `This is some weird stuff`:

```
*** =sct
```{python}
test_output_contains("This is some weird stuff", pattern = False)
success_msg("Great job!")
```
```

Notice that we set `pattern` to `False`, this will cause `test_output_contains()` to search for the pure string, no patterns are used. This SCT is not robust, because it won't be accepted if the student submits `print("This is some cool stuff")`, for example. Therefore, it's a good idea to use regular expressions. `pattern=True` by default, so there's no need to specify this:

```
*** =sct
```{python}
test_output_contains(/This is some \w* stuff/,
                     no_output_msg = "Print out `This is some ... stuff` to the␣
↪output, fill in `...` with a word you like.")
success_msg("Great job!")
```
```
```

```
```

Now, different printouts will be accepted. Notice that we also specified `no_output_msg` here. If the pattern is not found in the output generated, this message will be shown instead of a message that's automatically generated by `pythonwhat`.

## test_student_typed

**test_student_typed**(*text*, *pattern=True*, *not_typed_msg=None*, *state=None*)

Test the student code.

Tests if the student typed a (pattern of) text.

**Parameters**

- **text** (*str*) – the text that is searched for
- **pattern** (*bool*) – if True, the text is treated as a pattern. If False, it is treated as plain text. Defaults to False.
- **not_typed_msg** (*str*) – feedback message to be displayed if the student did not type the text.

```
test_student_typed(text,
                   pattern=True,
                   not_typed_msg=None)
```

`test_student_typed()` will look through the student's submission to find a match with the string specified in `text`. With `pattern`, you can declare whether or not to use regular expressions.

Suppose the solution of an exercise looks like this:

~~~
*** =solution
```{python}
# Calculate the sum of all single digit numbers and assign the result to 's'
s = sum(range(10))

# Print the result to the shell
print(s)
~~~
```

The following SCT tests whether the student typed `"sum(range("`:

```
*** =sct
```{python}
test_student_typed("sum(range(", pattern = False)
success_msg("Great job!")
```
```

Notice that we set `pattern` to `False`, this will cause `test_student_typed()` to search for the pure string, no patterns are used. This SCT is not that robust though, it won't accept something like `sum( range(10) )`. This is why we should almost always use regular expressions in `test_student_typed`. For example:

```
*** =sct
```{python}
test_student_typed("sum\s*\(\s*range\s*\(", not_typed_msg="You didn't use `range()`␣
↪inside `sum()`.")
success_msg("Great job!")
```
```
```

We also used `not_typed_msg` here, which will control the feedback given to the student when `test_student_typed()` doesn't pass. Note that also `success_msg()` is used here, this is the message that is shown when the SCT has passed.

In general, **you should avoid using `test_student_typed()`**, as it imposes severe restrictions on how a student can solve an exercise. Often, there are different ways to solve an exercise. Unless you have a very advanced regular expression, `test_student_typed()` will not be able to accept all these different approaches. For the example above, `test_function()` would be more appropriate.

## has_equal_ast

**`has_equal_ast`** (*incorrect_msg='FMT: Your code does not seem to match the solution.'*, *code=None*, *exact=True*, *state=None*)

Test whether abstract syntax trees match between the student and solution code.

**Parameters**

- **`incorrect_msg`** – message displayed when ASTs mismatch.
- **`code`** – optional code to use instead of the solution AST
- **`exact`** – whether the representations must match exactly. If false, the solution AST only needs to be contained within the student AST (similar to using test student typed).

**Example** Student and Solution Code:

```
dict(a = 'value').keys()
```

SCT:

```
# all pass
Ex().has_equal_ast()
Ex().has_equal_ast(code = "dict(a = 'value').keys()")
Ex().has_equal_ast(code = "dict(a = 'value')", exact = False)
```

An abstract syntax tree (AST) is a way of representing the high-level structure of python code.

### Example: quotes

Whether you use the concrete syntax `x = "1"` or `x = '1'`, the abstract syntax is the same: x is being assigned to the string "1".

### Example: parenthesis

Grouping by parentheses produces the same AST, when the same statement would work the same without them. For example, `(True or False) and True`, and `True or False and True`, are the same due to operator precedence.

### Example: spacing

The same holds for different types of spacing that essentially specify the same statement: `x = 1` or `x = 1`.

### Caveat: evaluating

What the AST doesn't represent is values that are found through evaluation. For example, the first item in the list in

```
x = 1
[x, 2, 3]
```

and

```
[1, 2, 3]
```

Is not the same. In the first case, the AST represents that a variable `x` needs to be evaluated in order to find out what its value is. In the second case, it just represents the value `1`.

## test_mc

**`test_mc`**(*correct*, *msgs*, *state=None*)

Test multiple choice exercise.

Test for a MultipleChoiceExercise. The correct answer (as an integer) and feedback messages are passed to this function.

**Parameters**

- **`correct`** (*`int`*) – the index of the correct answer (should be an instruction). Starts at 1.
- **`msgs`** (*`list (str)`*) – a list containing all feedback messages belonging to each choice of the
- **`The list should have the same length as the number of instructions.`** (*`student.`*) –

```
test_mc(correct, msgs)
```

Multiple choice exercises are straightforward to test. Use `test_mc()` to provide tailored feedback for both the incorrect options, as the correct option. Below is the code for a multiple choice exercise example, with an SCT that uses `test_mc`:

~~~
--- type:MultipleChoiceExercise lang:python xp:50 skills:2
## The author of Python

Who is the author of the Python programming language?

*** =instructions
- Roy Co
- Ronald McDonald
- Guido van Rossum

*** =hint
Just google it!

*** =sct
```{python}
test_mc(correct = 3,
        msgs = ["That's someone who makes soups.",
                "That's a clown who likes burgers.",
                "Correct! Head over to the next exercise!"])
```
~~~

The first argument of `test_mc()`, `correct`, should be the number of the correct answer in this list. Here, the correct answer is Guido van Rossum, corresponding to 3. The `msgs` argument should be a list of strings with a length equal to the number of options. We encourage you to provide feedback messages that are informative and tailored to the (incorrect) option that people selected. Make sure to correctly order the feedback message such that it corresponds to the possible answers that are listed in the instructions tab. Notice that there's no need for `success_msg()` in multiple choice exercises, as you have to specify the success message inside `test_mc()`, along with the feedback for incorrect options.

## Part Checks

### Check Syntax

#### In Brief

While functions beginning with `test_`, such as `test_student_typed` look over some code or output, `check_` functions allow us to zoom in on parts of that student and solution code.

For example, `check_list_comp` examines list comprehensions by breaking them into 3 parts: `body`, `comp_iter`, and `ifs`. This is shown below.

`[i*2 for i in range(10) if i>2]` => `[BODY for i in COMP_ITER if IFS]`

Each of these 3 parts may be tested individually using the simple test functions. For example, in order to test the body of the comprehension above, we could create the following exercise.

~~~
*** =solution
```{python}
L2 = [i**2 for i in range(0,10) if i>2]
```

*** =sct
```{python}
(Ex().check_list_comp(0)                        # focus on first list comp
       .check_body().test_student_typed('i\*2')  # focus on its body for test
       )
```
~~~

In the SCT, `check_list_comp` gets the first comprehension, and will fail with feedback if no comprehensions were used in th submission code. `check_body` gets `i**2` in the solution code, and whatever corresponds to BODY in the submission code.

(Note: the parentheses around the entire statement are just syntactic sugar, to let us chain commands in python without using \ at the end of each line.)

#### Full Example

This section expands the above example to run tests on each part: body, iter, and ifs.

~~~
*** =solution
```{python}
L2 = [i*2 for i in range(0,10) if i>2]
```

*** =sct
```{python}
~~~

```
list_comp = Ex().check_list_comp(0, missing_msg="Did you include a list comprehension?
↪")
list_comp.check_body().test_student_typed('i\*2')
list_comp.check_iter().has_equal_value()
list_comp.check_ifs(0).multi([has_equal_value(context_vals=[i]) for i in range(0,10)])
```
```

In this SCT, the first line focuses on the first list comprehension, and assigns it to `list_comp`, so we can test each part in turn. As a reminder, the code corresponding to each part in the solution code is..

- BODY: `i**2`
- COMP_ITER: `range(0,10)`
- IFS: `[i>2]`

Note that IFS is represented as a list, and the index 1 was passed to *check_ifs* because a list comprehension may have multiple if statements. Since the test on BODY, is explained in the [In Brief section](#In_Brief), we will focus on the tests on ITER and and IFS.

### check_iter

In the line `list_comp.check_iter().equal_value()`, `check_iter` gets the ITER part in the solution and submission code, while `has_equal_value` tells pythonwhat to run those parts and see if they return equal values. Below are example solution and submission codes, with the ITER part they would produce

| type | code | ITER part |
|---|---|---|
| **solution** | `[i*2 for i in range(0,10) if i>2]` | `range(0,10)` |
| **submission** | `[i*2 for i in range(10) if i>2]` | `range(10)` |

In this case, `equal_value` will run each part, and then confirm that `range(0,10) == range(10)`. For more on functions that run code, like `has_equal_value` see [Expressions Tests](processes).

### check_ifs

The line

```
list_comp.check_ifs(0).multi([has_equal_value(context_vals=[i] for i in range(0,10))])
```

is a doozy, but can be broken down into

```
equal_tests = [has_equal_value(context_vals=[i] for i in range(0,10))]     #␣
↪collection of has_equal_tests
list_comp.check_ifs(0).multi(equal_tests)                                   # focus on␣
↪IFS run equal_tests`
```

In this case `equal_tests` is a list of `has_equal_value` tests that we'll want to perform. `check_ifs(1)` grabs the first IFS part, and `multi(equal_tests)` runs each `has_equal_value` test on that part.

Notice that `has_equal_value` was given a context_val argument. This is because the list comprehension creates a temporary variable that needs to be defined when we run the IFS code.

| type | code | IFS part | context value |
|---|---|---|---|
| **solution** | `[i*2 for i in range(0,10) if i>2]` | `if i>2` | `i` |
| **submission** | `[j*2 for j in range(0,10) if j>2]` | `if j>2` | `j` |
```

In this case, the context_vals argument is a list of values, with one for each (in this case only a single) context value. In this way, `has_equal_value` assigns `i` and `j` to the same value, before running the IFs part. By creating a list of `has_equal_tests` with context vals spanning `range(0,10)`, we test the IFS across a range of values.

### Nested Part Example

Check functions may be combined to focus on parts within parts, such as

~~~
*** =solution
```{python}
[i*2 if i> 5 else 0 for i in range(0,10)]
~~~
```

In this case, a representation with the parts in caps and wrapping the inline if expression with `{BODY=...}` is

```
[{BODY=BODY if TEST else ORELSE} for i in ITER]
```

in order to test running the inline if expression we could go from list_comp => body => if_exp. One possible SCT is shown below.

```
*** =sct
```{python}
(Ex().check_list_comp(0)                        # first comprehension
     .check_body().set_context(i=6)       # comp's body
     .check_if_exp(0).has_equal_value()   # body's inline IFS
     )
```
```

Note that rather than using the `context_vals` argument of `has_equal_value` we use `set_context` to define the context variable (`i` in the solution code) on the body of the list comprehension. This makes it very clear when the context value was introduced. It is worth pointing out that of the parts a list comprehension has, BODY and IFS, but not ITER have `i` as a context value. This is because in python `i` is undefined in the ITER part. Context values are listed in the [see cheatsheet below].

### Testing only the body of the list comprehension

If we left out the `check_if_exp` above, the resulting SCT,

```
(Ex().check_list_comp(0).check_body().set_context(i=6)
         #.check_if_exp(1)
         .has_equal_value()
         )
```

would still run the same code for the solution (the inline if expression), since it's the only thing in the BODY of the list comprehension. However it wouldn't check if an if expression was used, allowing a wider range of passing and failing submissions (for better or worse!). Moreover, *has_equal_value* may be used multiple times during the chaining, as it doesn't change what the focus is.

## Helper Functions

### multi

**multi**(**args*, *state=None*)
Run multiple subtests. Return original state (for chaining).
```

### Comma separated arguments

For example, this code without multi,

```
Ex().check_if_exp(0).check_body().has_equal_value()
Ex().check_if_exp(0).check_test().has_equal_value()
```

is equivalent to

```
Ex().check_if_exp(0).multi(
        check_body().has_equal_value(),
        check_test().has_equal_value()
        )
```

### List or generator of subtests

Rather than one or more subtest args, multi can take a single list or generator of subtests. For example, the code below checks that the body of a list comprehension has equal value for 10 possible values of the iterator variable, `i`.

```
Ex().check_list_comp(0)
    .check_body()
    .multi(set_context(i=x).has_equal_value() for x in range(10))
```

### Chaining off multi

Multi returns the same state, or focus, it was given, so whatever comes after multi will run the same as if multi wasn't used. For example, the code below tests a list comprehension's body, followed by its iterator.

```
Ex().check_list_comp(0) \
    .multi(check_body().has_equal_value()) \
    .check_iter().has_equal_value()
```

### has_context

**`has_context`**(*incorrect_msg=None*, *exact_names=False*, *state=None*)

Tests whether context variables defined by the student match the solution, for a selected block of code. A context variable is one that is defined in a looping or block statement. For example, `ii` in the code below.

```
[ii + 1 for ii in range(3)]
```

By default, the test fails if the submission code does not have the same number of context variables. This is illustrated below.

**Solution Code**

```
# ii and ltr are context variables
for ii, ltr in enumerate(['a']): pass
```

**SCT**

```
Ex().check_for_loop(0).check_body().has_context()
```

#### Passing Submission

```
# still 2 variables, just different names
for jj, Ltr in enumerate(['a']): pass
```

#### Failing Submission

```
# only 1 variable
for ii in enumerate(['a']): pass
```

---

**Note:** that if you use `has_context(exact_names = True)`, then the submission must use the same names for the context variables, which would cause the passing submission above to fail.

---

### set_context

**set_context**(**args*, *state=None*, ***kwargs*)

Update context values for student and solution environments.

Note that excess args and unmatched kwargs will be unused in the student environment. If an argument is specified both by name and position args, will use named arg.

Sets the value of a temporary variable, such as `ii` in the list comprehension below.

```
[ii + 1 for ii in range(3)]
```

Variable names may be specified using positional or keyword arguments.

#### Example

#### Solution Code

```
ltrs = ['a', 'b']
for ii, ltr in enumerate(ltrs):
    print(ii)
```

#### SCT

```
Ex().check_for_loop(0).check_body() \
    .set_context(ii=0, ltr='a').has_equal_output() \
    .set_context(ii=1, ltr='b').has_equal_output()
```

Note that if a student replaced *ii* with *jj* in their submission, *set_context* would still work. It uses the solution code as a reference. While we specified the target variables `ii` and `ltr` by name in the SCT above, they may also be given by position..

```
Ex().check_for_loop(0).check_body().set_context(0, 'a').has_equal_output()
```

#### Instructor Errors

If you are unsure what variables can be set, it's often easiest to take a guess. When you try to set context values that don't match any target variables in the solution code, `set_context` raises an exception that lists the ones available.

### with_context

**`with_context`**(**args*, *state=None*)

Runs subtests after setting the context for a `with` statement.

This function takes arguments in the same form as `multi`. Note also that `with_context` was the default behavior for `test_with` in pythonwhat version 1.

#### Context Managers Explained

With statements are special in python in that they enter objects called a context manager at the beginning of the block, and exit them at the end. For example, the object returned by `open('fname.txt')` below is a context manager.

```
with open('fname.txt') as f:
    print(f.read())
```

This code runs by

1. assigning `f` to the context manager returned by `open('fname.txt')`
2. calling `f.__enter__()`
3. running the block
4. calling `f.__exit__()`

`with_context` was designed to emulate this sequence of events, by setting up context values as in step (1), and replacing step (3) with any sub-tests given as arguments.

### fail

**`fail`**(*msg=''*, *state=None*)
Fail test with message

Fails. This function takes a single argument, `msg`, that is the feedback given to the student. Note that this would be a terrible idea for grading submissions, but may be useful while writing SCTs. For example, failing a test will highlight the code as if the previous test/check had failed.

As a trivial SCT example,

```
Ex().check_for_loop(0).check_body().fail()      # fails boo
```

This can also be helpful for debugging SCTs, as it can be used to stop testing as a given point.

## Check Functions

**Arguments**

- **index**: index or key corresponding to the node or part of interest. This applies to all functions in the **check** column in the table below. However, apart from that, it only applies when there is more than one of a specific part to choose from — `check_ifs`, `check_args`, `check_handlers`, and `check_context`. (e.g. `Ex().check_list_comp(0).check_ifs(0)`)
- **missing_msg**: optional feedback message if node or part doesn't exist.

Note that code in all caps indicates the name of a piece of code that may be inspected using, `check_{part}`, where `{part}` is replaced by the name in caps (e.g. `check_if_else(0).check_test()`). Target variables are those that may be set using `set_context`. These variables may only be set in places where python would set them. For example, this means that a list comprehension's ITER part has no target variables, but its BODY does.

| check | parts | target variables |
|---|---|---|
| check_if_else(0) | `if TEST:` `    BODY` `else:` `    ORELSE` | |
| check_while(0) | `while TEST:` `    BODY` `else:` `    ORELSE` | |
| check_list_comp(0) | `[BODY for i in ITER if IFS[0] if IFS[1]]` | `i` |
| check_generator_exp(0) | `(BODY for i in ITER if IFS[0] if IFS[1])` | `i` |
| check_dict_comp(0) | `{KEY : VALUE for k, v in ITER if IFS[0]}` | `k, v` |
| check_for_loop(0) | `for i in ITER:` `    BODY` `else:` `    ORELSE` | `i` |
| check_try_except(0) | `try:` `    BODY` `except BaseException as e:` `    HANDLERS['BaseException']` `except:` `    HANDLERS['all']` `else:` `    ORELSE` `finally:` `    FINALBODY` | `e` |
| check_with(0) | `with CONTEXT[0] as f1, CONTEXT[1] as f2:` `    BODY` | `f` |
| check_function_def('f') | `def f(ARGS[0], ARGS[1]):` `    BODY` | argument names |
| check_lambda(0) | `lambda ARGS[0], ARGS[1]: BODY` | argument names |
| check_function('f', 0) | `f(ARGS[0], ARGS[1])` | argument names |

## More

### elif statements

In python, when an if-else statement has an elif clause, it is held in the ORELSE part,

```
if TEST:
    BODY
ORELSE          # elif and else portion
```

In this sense, an if-elif-else statement is represented by python as nested if-elses. For example, the final `else` below

```
if x:    print(x)         # line 1
elif y: print(y)          #   ""   2
else:    print('none')    #   ""   3
```

can be checked with the following SCT

```
(Ex().check_if_else(0)                        # lines 1-3
     .check_orelse().check_if_else(0)      # lines 2-3
     .check_orelse().has_equal_output()        # line 3
     )
```

### function definition / lambda args

the ARGS part in function definitions and lambdas may be selected by position or keyword. For example, the arguments *a* and *b* below,

```
def f(a, b=2, *some_name):
    BODY
```

Could be tested using,

```
Ex().check_function_def('f').multi(
        check_args('a').is_default(),
        check_args('b').is_default().has_equal_value(),
        check_args('*args', 'missing a starred argument!')
        )
```

Note that `check_args('*args')` and `check_args('**kwargs')` may be used to test *args, and **kwargs style parameters, regardless of their name in the function definition.

### function call args

Behind the scenes, `check_function` uses the same logic for matching arguments to function signatures as test_function_v2. It also has a `signature` argument that accepts a custom signature.

### Matching Signatures

By default, `check_function` tries to match each argument in the function call with the appropriate parameters in that function's call signature. For example, all the calls to `f` below use `a = 1` and `b = 2`.

```python
def f(a, b): pass

f(1, 2)             # by position
f(a = 1, b = 2)     # by keyword
f(1, b = 2)         # mixed
```

However, when testing a submission, we may not care how the argument was specified.

~~~
*** =pre_exercise_code
```{python}
def f(a, b): pass
```

*** =solution
```{python}
f(1, b=2)
```

*** =sct
```{python}
Ex().check_function('f', 0).check_args('a').has_equal_value()
```
~~~

will pass for all the ways of calling `f` listed above.

### signature = False

Setting signature to false, as below, only allows you to check an argument by name, if the name was explicitly specified in the function call. For example,

~~~
*** =solution
```{python}
dict( [('a', 1)],  c = 2)
```

*** =sct
```{python}
Ex().check_function('dict', 0, signature=False)\
    .multi(
        check_args(0),      # can only select by position
        check_args('c')     # could use check_args(1)
        )
```
~~~

Note that here, an argument's position is referring to its position in the function call (not its signature).

### Example: testing a list passed as an argument

Suppose you want to test the first argument passed to *sum*. Below, we show how this can be down, using *has_equal_ast()* to check that the abstract syntax trees for the 1st argument match.

~~~
*** =solution
```{python}
sum([1, 2, 3])
```
~~~

~~~
*** =sct
```{python}
(Ex().check_function('sum', 0)
     .check_args(0)
     .has_equal_ast("ast fail")                          # compares abstract␣
↪representations
     .test_student_typed("\[1, 2, 3\]", "typed fail")  # alternative, more rigid test
     )
~~~
```

Notice that testing the argument is similar to testing, say, the body of an if statement. In this sense, we could even do deeper checks into an argument. Below, the SCT verifies that the first argument passed to sum is a list comprehension.

```
*** =solution
```{python}
sum([i for i in range(10)])
```

*** =sct
```{python}
(Ex().check_function('sum', 0)
     .check_args(0)
     .check_list_comp(0)
     .has_equal_ast()
     )
```
```

## Expressions

Expression tests run pieces of the student and solution code, and then check the resulting value, printed output, or errors they produce.

### `has_equal` Syntax

Once student/submission code has been selected using a check test, we can run it using one of three functions. They all take the same arguments, and run the student and submission code in the same way. However, they differ in how they compare the outcome:

- has_equal_value - compares the value returned by the code.
- has_equal_output - compares printed output.
- has_equal_error - compares any errors raised.

**`has_expr`**(*incorrect_msg='FMT:Unexpected expression {test}: expected '{sol_eval}', got '{stu_eval}' with values{extra_env}.', error_msg='Running an expression in the student process caused an issue.', undefined_msg='FMT:Have you defined '{name}' without errors?', extra_env=None, context_vals=None, expr_code=None, pre_code=None, keep_objs_in_env=None, name=None, highlight=None, state=None, test=None*)

Run student and solution code, compare returned value, printed output, or errors.

**Parameters**
```

- **incorrect_msg** (*str*) – feedback message if the output of the expression in the solution doesn't match the one of the student. This feedback message will be expanded if it is used in the context of another test function, like test_if_else.
- **error_msg** (*str*) – feedback message if there was an error when running the student code. Note that when testing for an error, this message is displayed when none is raised.
- **undefined_msg** (*str*) – feedback message if the name argument is defined, but a variable with that name doesn't exist after running the student code.
- **extra_env** (*dict*) – set variables to the extra environment. They will update the student and solution environment in the active state before the student/solution code in the active state is ran. This argument should contain a dictionary with the keys the names of the variables you want to set, and the values are the values of these variables.
- **context_vals** (*list*) – set variables which are bound in a for loop to certain values. This argument is only useful if you use the function in a test_for_loop. It contains a list with the values of the bound variables.
- **expr_code** (*str*) – if this variable is not None, the expression in the student/solution code will not be ran. Instead, the given piece of code will be ran in the student as well as the solution environment and the result will be compared.
- **pre_code** (*str*) – the code in string form that should be executed before the expression is executed. This is the ideal place to set a random seed, for example.
- **keep_obj_in_env** (*list ()*) – a list of variable names that should be hold in the copied environment where the expression is evaluated. All primitive types are copied automatically, other objects have to be passed explicitely.
- **name** (*str*) – the name of a variable, or expression, whose value will be tested after running the student and solution code. This could be thought of as post code.

### Basic Usage

#### Running the whole code submission

In the example below, we re-run the entire student and submission code, and check that they print out the same output.

~~~
*** =solution
```{python}
x = [1,2,3]
print(x)
```

*** =sct
```{python}
# run all code and compare output
Ex().has_equal_output()
# equivalent to
# Ex().test_output_contains('[1,2,3]')
```
~~~

Note that while we could have used `test_output_contains` to verify that the student printed `"[1, 2, 3]"`, using `has_equal_output` simply requires that the student output matches the solution output.

### Running part of the code

Combining an expression test with part checks will run only a piece of the submitted code. The example below first uses `has_equal_value` to run an entire if expression, and then to run only its body.

~~~
*** =solution
```{python}
x = [1,2,3]
sum(x) if x else None
```

*** =sct
```{python}
# test body of if expression
(Ex().check_if_exp(0)     # focus on if expression
     .has_equal_value()   # run entire if expression, check value
     .check_body()        # focus on body "sum(x)"
     .has_equal_value()   # run body, check value
     )
```
~~~

Note that commands chaining off of `has_equal_value` behave as they would have if `has_equal_value` weren't used. In this sense, the `check_body` behaves the same in

```
Ex().check_if_exp(0).has_equal_value().check_body()
```

and

```
Ex().check_if_exp(0).check_body()
```

in that it gets "sum(x)" in the solution code (and its corresponding code in the submission).

### Context Values

Suppose we want the student to define a function, that loops over the elements in a dictionary, and prints out each key and value, as follows:

~~~
*** =solution
```{python}
def print_dict(my_dict):
    for key, value in my_dict.items():
        print(key + " - " + str(value))
```
~~~

An appropriate SCT for this exercise could be the following (for clarity, we're not using any default messages):

~~~
*** =sct
```{python}
# get for loop code, set context for my_dict argument
for_loop = (Ex()
     .check_function_def('print_dict')          # ensure 'print_dict' is defined
     .check_body()                               # get student/solution code in body
     .set_context(my_dict = {'a': 2, 'b': 3})   # set print_dict's my_dict arg
     .check_for_loop(0)                          # ensure for loop is defined
     )

# test for loop iterator
~~~

```
for_loop.check_iter().has_equal_value()          # run iterator (my_dict.items())
# test for loop body
for_loop.check_body().set_context(key = 'c', value = 3).has_equal_value()

```
```

Assuming the student coded the function in the exact same way as the solution, the following things happen:

- checks whether `print_dict` is defined, then gets the code for the function definition body.
- because `print_dict` takes an argument `my_dict`, which would be undefined if we ran the body code, `set_context` defines what `my_dict` should be when running the code. Note that its okay if the submitted code named the argument `my_dict` something else, since set_context matches submission / solution arguments up by position.

When running the bottom two SCTs for the for_loop

- `for_loop.check_iter().has_equal_value()` - runs the code for the iterator, `my_dict.items()` in the solution and its corresponding code in the submission, and compares the values they return.
- `for_loop.check_body().set_context(key = 'c', value = 3).has_equal_value()` - runs the code in the for loop body, `print(key + " - " + str(value))` in the solution, and compares outputs. Since this code may use variables the for loop defined, `key` and `value`, we need to define them using `set_context`.

### How are Context Values Matched?

Context values are matched by position. For example, the submission and solution codes...

*** =solution

```
for ii, x in enumerate(range(3)): print(ii)
```

*** =submission

```
for jj, y in enumerate(range(3)): print(jj)
```

Using `Ex().check_for_loop(0).check_body().set_context(...)` will do the following...

| statement | solution (ii, x) | submission (jj, y) |
|---|---|---|
| set_context(ii=1, x=2) | ii = 1, x = 2 | jj = 1, y = 2 |
| set_context(ii=1) | ii = 1, x is undefined | jj = 1, y is undefined |
| set_context(x=2) | ii is undefined, x = 2 | jj is undefined, y = 2 |

**Note:** If ::set_context:: does not define a variable, nothing is done with it. This means that in the code examples above, running the body of the for loop would call print with ::ii:: or ::jj:: left at 2 (the values they have in the solution/submission environments).

### pre_code: fixing mutations

Python code commonly mutates, or changes values within an object. For example, the variable `x` points to an object that is mutated every time a function is called.

```
x = {'a': 1}

def f(d): d['a'] += 1
```
```

```
f(x)      # x['a'] == 2 now
f(x)      # x['a'] == 3 now
```

In this case, when `f` is run, it changes the contents of `x` as a side-effect and returns None. When using SCTs that run expressions, mutations in either the solution or submission environment can cause very confusing results. For example, calling `np.random.random()` will advance numpy's random number generator. In the code below the random seed is set to 42, but the solution code advances the random generator further than the submission code. As a result the SCT will fail.

~~~
*** =pre_exercise_code
```{python}
import numpy as np
np.random.seed(42)                # set random generator seed to 42
```

*** =solution
```{python}
if True: np.random.random()       # 1st random call: .37

np.random.random()                # 2nd random call: .95
```

*** =submission
```{python}
if True: np.random.random()       # 1st random call: .37

# forgot 2nd call to np.random.random()
```

*** =sct
```{python}
# Should pass but fails, because random generator has advanced
# twice in solution, but only once in submission
Ex().check_if_else(0).check_body().has_equal_value()
```
~~~

In order to test random code, the random generator needs to be at the same state between submission and solution environments. Since their generators can be thrown out of sync, the most reliable way to do this is to set the seed using the `pre_code` argument to `has_equal_value`. In the case above, the sct may be fixed as follows

*** =sct

```
Ex().check_if_else(0).check_body().has_equal_value(pre_code = "np.random.seed(42)")
```

More generally, it can be helpful to define a pre_code variable to use before expression tests...

*** =sct

```
pre_code = """
np.random.seed(42)
"""

Ex().has_equal_output(pre_code=pre_code)
Ex().check_if_else(0).check_body().has_equal_value(pre_code = pre_code)
```

### extra_env: fixing slow SCTs

The `extra_env` argument is similar to `pre_code`, in that you can (re)define objects in the student and submission environment before running an expression. The difference is that, rather than passing a string that is executed in each environment, extra_env lets you pass objects directly. For example, the two SCTs below are equivalent...

*** =sct

```
Ex().has_equal_value(pre_code="x = 10")
Ex().has_equal_value(extra_env = {'x': 10})
```

In practice they can often be used interchangably. However, one area where `extra_env` may shine is in mocking up data objects before running tests. For example, if the SCT below didn't use extra_env, then it would take a long time to run.

*** =pre_exercise_code

```
a_list = list(range(10000000))
```

*** =solution

```
print(a_list[1])
```

*** =sct

```
extra_env = {'a_list': list(range(10))}
Ex().has_equal_output(extra_env = extra_env)
```

The reason extra_env is important here, is that pythonwhat tries to make a deepcopy of lists, so that course developers don't get bit by unexpected mutations. However, the larger the list, the longer it takes to make a deepcopy. If an SCT is running slowly, there's a good chance it uses a very large object that is being copied for every expression test.

### name: run tests after expression

### expr_code: change expression

The `expr_code` argument takes a string, and uses it to replace the code that would be run by an expression test. For example, the following SCT simply runs `len(x)` in the solution and student environments.

~~~
*** =solution
```{python}
# keep x the same length
x = [1,2,3]
~~~

*** =SCT
```{python}
Ex().has_equal_value(expr_code="len(x)")
```
```

**Note:** Using *expr_code* does not change how expression tests perform highlighting. This means that *Ex().for_loop(0).has_equal_value(expr_code="x[0]")* would highlight the body of the checked for loop.
```

### `call` Syntax

Testing a function definition or lambda may require calling it with some arguments. In order to do this, use the `call()` SCT. There are two ways to tell it what arguments to pass to the function/lambda,

- `call("f (1, 2, x = 3)")` - as a string, where `"f"` gets substituted with the function's name.
- `call([1,2,3])` - as a list of positional arguments.

Below, two alternative ways of specifying the arguments to pass are shown.

~~~
*** =solution
```{python}
def my_fun(x, y = 4, z = ('a', 'b'), *args, **kwargs):
    return [x, y, *z, *args]
~~~

*** =sct
```{python}
Ex().check_function_def('my_fun').call("f(1, 2, (3,4), 5, kw_arg='ok')")  # as string
Ex().check_function_def('my_fun').call([1, 2, (3,4), 5])                  # as list
```
```

**Note:** Technically, you can get crazy and replace the list approach with a dictionary of the form `{'args': [POSARG1, POSARG2], 'kwargs': {KWARGS}}`.

#### Additional Parameters

In addition to its first argument, `call()` accepts all the parameters that the expression tests above can (i.e. `has_equal_value`, `has_equal_error`, `has_equal_output`). The function call is run at the point where these functions would evaluate an expression. Moreover, setting the argument `test` to either "value", "output", or "error" controls which expression test it behaves like.

For example, the SCT below shows how to run some `pre_code`, and then evaluate the output of a call.

```
Ex().check_function_def('my_fun').call("f(1, 2)", test="output", pre_code="x = 1")
```

### Managing Processes

As mentioned on the *Homepage*, DataCamp uses two separate processes. One process to run the solution code, and one process to run the student's submission. This way, `pythonwhat` has access to the 'ideal ending scenario' of an exercises; this makes it easier to write SCTs. Instead of having to specify which value an object should be, we can have `test_object()` look into the solution process and compare the object in that process with the object in the student process.

#### Problem

Fetching Python objects or the results of running expressions inside a process is not straightforward. To be able to pull data from a process, Python needs to 'pickle' and 'unpickle' files: it converts the Python objects to a byte representation (pickling) that can be passed between processes, and then, inside the process that you want to work with the object, builds up the object from the byte representation again (unpickling).
```

For the majority of Python objects, this conversion to and from a byte representation works fine, but for some objects, it doesn't. Even `dill`, and improved implementation of `pickle` that's being used in `pythonwhat`, doesn't flawlessly convert all Python objects out there.

If you're writing an SCT with functions that require work in the solution process, such as `test_object()`, `test_function()`, and `test_function_definition()`, and then upload the exercise and test it on DataCamp, that you get backend errors that look like this:

```
... dilling inside process failed - write manual converter
... undilling of bytestream failed - write manual converter
```

The first error tells you that 'dilling' - or 'pickling', converting the object to a bytestream representation, failed. The second error tells you that 'undilling' - or 'unpickling', converting the byte representation back to a Python object, failed. These errors will typically occur if you're dealing with exotic objects, such as objects that interface to files, connections to databases, etc.

### Solution

To be able to handle these errors, `pythonwhat` allows you to write your own converters for Python objects. Say, for example, that you're writing an exercise to import Excel data into Python, and you're using the `pandas` package. This is the solution and the corresponding SCT:

~~~
*** =solution
```{python}
import pandas as pd
xl = pd.ExcelFile('battledeath.xlsx')
```

*** =sct
```{python}
Ex().test_object('xl')
```
~~~

Suppose now that objects such as `xl`, which are of the type `pandas.io.excel.ExcelFile`, can't be properly dilled and undilled. (Note: because of hardcoded converters inside `pythonwhat`, they can, see below). To make sure that you can still use `test_object('xl')` to test the equality of the `xl` object between student and solution process, you can manually define a converter with the `set_converter()` function. You can extend the SCT as follows:

~~~
*** =sct
```
def my_converter(x):
    return(x.sheet_names)
set_converter(key = "pandas.io.excel.ExcelFile", fundef = my_converter)
Ex().test_object('xl')
```
~~~

With a lambda function, it's even easier:

~~~
*** =sct
```
set_converter(key = "pandas.io.excel.ExcelFile", fundef = lambda x: x.sheet_names)
Ex().test_object('xl')
```
~~~

The first arguemnt of `set_converter()`, the `key` takes the type of the object you want to add a manual converter for as a string. The second argument, `fundef`, takes a function definition, taking one argument and returning a single object. This function definition converts the exotic object into something more standard. In this case, the function converts the object of type `pandas.io.excel.ExcelFile` into a simple list of strings. A list of strings is something that can easily be converted into a bytestream and back into a Python object again, hence solving the problem.

If you want to reuse the same manual converter over different exercises, you'll have to use `set_converter()` in every SCT.

### Hardcoded converters

Some converters will be required often. For example, the result of calling `.keys()` and `.items()` on dictionaries can't be dilled and undilled without extra work. To handle these common yet problematic situations, `pythonwhat` features a list of hardcoded converters. This list is available in the source code; feel free to do a pull request if you want to add more converts to this list. This will reduce the amount of code duplication you have to do if you want to reuse the same converter in different exercises.

### Custom Equality

The `set_converter()` function opens up possibilities for objects that can actually be dilled and undilled perfectly fine. Say you want to test a `numpy` array, but you only want to check only if the dimensions of the array the student codes up match those in the solution process. You can easily write a manual converter that overrides the typical dilling and undilling of Numpy arrays, implementing your custom equality behavior:

~~~
*** =solution
```{python}
import numpy as np
my_array = np.array([[1,2], [3,4], [5,6]])
```

*** =sct
```
set_converter(key = "numpy.ndarray", fundef = lambda x: x.shape)
Ex().test_object('my_array')
```
~~~

Both of the following submissions will be accepted by this SCT:

- `my_array = np.array([[1,2], [3,4], [5,6]])`
- `my_array = np.array([[0,0], [0,0], [5,6]])`

## Logic Tests

### test_correct

**`test_correct`**(*check*, *diagnose*, *state=None*)
Allows feedback from a diagnostic SCT, only if a check SCT fails.

A wrapper function around `test_or()`, `test_correct()` allows you to add logic to your SCT. Normally, your SCT is simply a script with subsequent `pythonwhat` function calls, all of which have to pass. `test_correct()` allows you to bypass this: you can specify a "sub-SCT" in the `check` part, that should pass. If these tests pass, the "sub-SCT" in `diagnose` is not executed. If the tests don't pass, the "sub-SCT" in `diagnose` is run, typically to dive deeper into what the error might be and give more specific feedback.

To accomplish this, the SCT in `check` is executed silently, so that failure will not cause the SCT to stop and generate a feedback message. If the execution passes, all is good and `test_correct()` is abandoned. If it fails, `diagnose` is executed, not silently. If the `diagnose` part fails, the feedback message that it generates is presented to the student. If it passes, the `check` part is executed again, this time not silently, to make sure that a `test_correct()` that contains a failing `check` part leads to a failing SCT.

### Example 1

As an example, suppose you want the student to calculate the mean of a Numpy array `arr` and store it in `res`. A possible solution could be:

~~~
*** =solution
```{python}
# Import numpy and create array
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])

# Calculate result
result = np.mean(arr)
~~~
```

You want the SCT to pass when the student manages to store the correct value in the object `result`. How `result` was calculated, does not matter to you: as long as `result` is correct, the SCT should accept the submission. If something about `result` is not correct, you want to dig a little deeper and see if the student used the `np.mean()` function correctly. The following SCT will do just that:

```
*** =sct
```{python}
test_correct(test_object('result'),
             test_function('numpy.mean'))
success_msg("You own numpy!")
```
```

Let's go over what happens when the student submits different pieces of code:

- The student submits `result = np.mean(arr)`, exactly the same as the solution. `test_correct()` runs `test_object('result')`. This test passes, so `test_correct()` stops. The SCT passes.
- The student submits `result = np.sum(arr) / arr.size`, which also leads to the correct value in `result`. `test_correct()` runs `test_object('result')`. This test passes, so `test_correct()` stops before running `test_function()`. The entire SCT passes even though `np.mean()` was not used.
- The student submits `result = np.mean(arr + 1)`. `test_correct()` runs `test_object('result')`. This test fails, so `test_correct()` continues with 'diagnose', running `test_function('numpy.mean')`. This function fails, since the argument passed to `numpy.mean()` in the student submission does not correspond to the argument passed in the solution. A meaningful, specific feedback message is presented to the student: you did not correctly specify the arguments inside `np.mean()`.
- The student submits `result = np.mean(arr) + 1`. `test_correct()` runs `test_object('result')`. This test fails, so `test_correct()` continues with'diagnose', running `test_function('numpy.mean'). This function passes, because`np.mean()`is called in exactly the same way in the student code as in the solution. Because there is something wrong –`result`is not correct – the 'check' SCT, `test_object('result')`is executed again, and this time its feedback on failure is presented to the student. The student gets the message that`result' does not contain the correct value.
```

### Multiple functions in `diagnose` and `check`

You can also use `test_correct()` with entire 'sub-SCTs' that are composed of several SCT calls. In this case, you may put multiple tests inside `multi()`, as below..

~~~
*** =sct
```{python}
Ex().test_correct(
        multi(test_object('a'), test_object('b')),   # multiple check SCTs
        test_function('numpy.mean')
        )
~~~
```

### Why to use `test_correct()`

You will find that `test_correct()` is an extremely powerful function to allow for different ways of solving the same problem. You can use `test_correct()` to check the end result of a calculation. If the end result is correct, you can go ahead and accept the entire exercise. If the end result is incorrect, you can use the `diagnose` part of `test_correct()` to dig a little deeper.

It is also perfectly possible to use `test_correct()` inside another `test_correct()`.

### Wrapper around `test_or()`

`test_correct()` is a wrapper around `test_or()`. `test_correct(diagnose, check)` is equivalent with:

```
def diagnose_and_check()
    diagnose()
    check()

test_or(diagnose_and_check, check)
```

Note that in each of the `test_or` cases here, the submission has to pass the SCTs specified in `check`.

## test_or

**`test_or`**(*\*tests*, *state=None*)
    Test whether at least one SCT passes.

This function simply tests whether one of the SCTs you specify inside it passes.

Suppose you want to check whether people correctly printed out any integer between 3 and 7. A solution could be:

```
    *** =solution
    ```{python}
    print(4)
    ```
```

To test this in a robust way, you could use `test_output_contains()` with a suitable regular expression that covers everything, or you can use `test_or()` with three separate `test_output_contains()` functions.

```
    *** =sct
    ```{python}
    test_or(test_output_contains('4'),
```

```
            test_output_contains('5'),
            test_output_contains('6'))
    success_msg("Nice job!")
    ```
```

You can consider `test_or()` a logic-inducing function. The different calls to `pythonwhat` functions that are in your SCT are actually all tests that *have* to pass: they are `AND` tests. With `test_or()` you can add chunks of `OR` tests in there.

## Spec2 Changes

### Lambdaless test functions

Sometimes you want to pass a test function as an argument to another test function. For the examples below, we'll use the following solution code

```
# solution code
[1 if True else 0 for i in range(10)]
```

#### Removing Lambdas

Using an SCT that works in pythonwhat v2,

```
# pythonwhat v2 SCT
test_list_comp(
        body=test_if_exp(
                body=test_student_typed('1'))       # line 4
```

would do the following (in order)...

1. run the test_student_typed function on line 4, which runs over the whole code, rather than just the body of the inline if.
2. pass its return value (None) to the body argument of `test_if_exp`.
3. run `test_if_exp`, whose body argument is None, rather than a sub-test.
4. run test_list_comp, whose body argument is None, rather than a sub-test.

Instead, in pythonwhat v1, testing the inline if expression (*1 if True else 0*) requires an SCT peppered with lambdas, such as

```
# v1 SCT
test_list_comp(
        body=lambda: test_if_exp(
                body = lambda: test_student_typed('1'))
```

#### Removing temporary functions for multiple sub-tests

In pythonwhat v2, multiple sub-tests may be run by putting them in a list, such as
```

```
# v2 SCT
# list of sub-tests
if_body_tests = [test_student_typed('1'), test_expression_result(context_vals=[1])]
# main test
test_list_comp(
        body = test_if_exp(
                  body = if_body_tests)    # list could also be put here directly
```

which, in pythonwhat v1 would try to run the tests in `if_body_tests` first, rather than as sub-tests. In order to accomplish this in pythonwhat v1, temporary functions were necessary, such as,

```
# v1 SCT
# temporary function for testing inline if expression
def inner_test():
    test_student_typed('1')
    test_expression_result(context_vals=[1])
# main test
test_list_comp(
        body=lambda: test_if_exp(body = inner_test))
```

while not too different, this approach can spiral out of control for complex SCTs (temporary functions within temporary functions, etc..).

### How it works

| spec | SCT | effect |
|---|---|---|
| v1 test | `test_list_comp()` | runs test |
| v1 test | `lambda: test_list_comp()` | waits to run |
| v2 check | `check_list_comp()` | waits to run |
| v2 check | `Ex().check_list_comp()` | runs test |
| v2 test | `F().test_list_comp()` | waits to run |
| v2 test | `test_list_comp()` | runs test if not argument to another |

The critical message is in pythonwhat

- **v1**: you have to do something special (use a lambda) to **opt-out** of running a test immediately.
- **v2**: you have to do something special (use `Ex()`) to **opt-in** to running a test immediately.

### pythonwhat v2 is Backwards Compatibile

for all test_ functions, pythonwhat v2's behavior is completely backwards compatible (and in fact was put in pythonwhat v1 several weeks before releasing v2). If you want to be explicit about any test function not being run, you can use the function chain object `F`, for example

```
# Implicit
sub_test = test_if_exp(ETC...)          # waits to run only if passed to another SCT
test_list_comp(body=sub_test)           # comment out this line, and sub_test will␣
↪run (as in pythonwhat v1)

# Explicit
sub_test = F().test_if_exp(ETC...)      # always waits to run
Ex().test_list_comp(body=sub_test)
```

### Never mix Explicit and Implicit approaches

If you choose to use the explicit approach (`Ex()` and `F()`), **don't expect the implicit approach to work**. That is, if you want `test_if_exp` below to run immediately, do not write

```
test_if_exp(1)              # implicit, should use Ex() or F()
Ex().check_list_comp(1)     # explicit
```

and expect the SCTs to run in a predictable order.

If you want to create a bunch of sub-tests, but don't want to preface each with F(), you can use the pythonwhat v2 function multi, as below.

```
subtest = multi(test_if_exp(ETC...), test_list_comp(ETC...))
```

## Context values for nested parts

Context values may now be defined for nested parts. For example, the print statement below,

```
for i in range(2):                  # outer for loop part
    for j in range(3):              # inner for loop part
        print(i + j)
```

may be tested by setting context values at each level,

```
(Ex()
    .check_for_loop(0).check_body().set_context(i = 1)     # outer for
    .check_for_loop(0).check_body().set_context(j = 2)     # inner for
        .has_equal_output()
    )
```

For more on context valus see [PROCESSES LINK HERE].

## Can call code chunks that before could only be split up

Entire code pieces, such as the inline if statement below,

```
'yes' if True else 'no'
```

may be tested using something like,

```
Ex().check_if_exp(0).has_equal_value()
```

## Argument checking

The arguments of a function definition, such as

```
def f(a=1): print(a)
```

are now parts and may be checked as below..

```
(Ex().check_function_def('f')    # does f exist?
     .check_args('a')            # does a exist?
     .is_default()               # is it a default argument?
     .has_equal_value()          # is it's default equal to solution?
     )
```

For more on the argument part, see [PART CHEATSHEET LINK HERE].

### Deprecate test_expression_result and friends

In pythonwhat v1, the functions

- test_expression_result
- test_expression_output
- test_object_after_expression

and various arguments of test_function_definition, test_lambda_function ran code and then checked the result, printed output, or errors against eachother.

These functions have been deprecated in favor of similar function..

- *has_equal_value*
- *has_equal_output*
- *has_equal_error*

These functions include identical arguments as the above.

### Feedback messages may use templating (via str.format or Jinja2)

**This feature is not stable, and should not be used in production**

### Cleaned up internals

Yayyyy.

# CHAPTER 2

# Pythonwhat V1

## Legacy Tests

The functions below combine both part checks and simple tests from pythonwhat v1. In some cases, they allow very specific checks that are not yet exposed to SCT creators in v2 (such as whether iterator variables have the exact same names).

### test_data_frame

```
def test_data_frame(name,
                    columns=None,
                    undefined_msg=None,
                    not_data_frame_msg=None,
                    undefined_cols_msg=None,
                    incorrect_msg=None)
```

Test a pandas DataFrame. This methods makes it possible to test the columns of a DataFrame object independently. Only the contents will be tested. Customisable error messages are possible for when there is no object in the process with name `name`, for when that object is no pandas DataFrame, when there are columns you want to test for which are not defined and when some columns contain bad values. `columns` contains a list of column names, and defaults to `None`. If it's `None`, all columns that are found in the data frame created by the solution will be tested.

#### Example 1

Suppose we have the following solution:

~~~
*** =solution
```{python}
# import pandas
import pandas as pd
~~~

```
# Create dataframe with columns a and b
my_df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5,  6]})
```
```

To test this we simply use:

```
*** =sct
```{python}
test_import("pandas")
test_data_frame("my_df", columns = ["a", "b"])
success_msg("Great job!")
```
```

This SCT will first test if `pandas` is correctly imported, and will then check if the student created a Pandas DataFrame called `my_df`. If it was not defined, a message is generated that you can override with `undefined_msg`. If the object was defined but it isn't a Pandas DataFrame, as message is generated that you can override wiht `not_data_frame_msg`. If `my_df` is a Pandas DataFrame, `test_data_frame()` goes on to check if all columns that are specified in the `columns` argument are defined in the data frame, and next whether these columns are correct. The messages that are generated in case of an incorrect submission can be overrided with `undefined_cols_msg` and `incorrect_msg`, respectively.

**NOTE**: Behind the scenes, `pythonwhat` has to fetch the value of objects from sub-processes. The required 'dilling' and 'undilling' can cause issues for exotic objects. For more information on this and possible errors that can occur, read the Processes article.

## test_dictionary

**test_dictionary**(*name*, *keys=None*, *undefined_msg=None*, *not_dictionary_msg=None*, *key_missing_msg=None*, *incorrect_value_msg=None*, *state=None*)

Test the contents of a dictionary.

```
def test_dictionary(name,
                    keys=None,
                    undefined_msg=None,
                    not_dictionary_msg=None,
                    key_missing_msg=None,
                    incorrect_value_msg=None)
```

Test a dictionary. Consider this function an advanced version of `test_object`, where you can specify messages that are explicit to test dictionaries. `test_dictionary` takes a step-by-step approach to checking the correspondence of the dictionary between student and solution process:

- Step 1: Is the object specified in `name` actually defined?
- Step 2: Is the object specified in `name` actually a dictionary?
- Step 3: For each key, is the key specified in the dictionary?
- Step 4: For each key, is the value corresponding to the key correct when comparing to the solution?

For Step 3 and Step 4, you can control which keys have to be tested through the `keys` argument. If you don't specify this argument, `test_dictionary()` will look for all keys and compare the values that are specified in the corresponding dictionary in the solution process.
```

### Example: step by step

Suppose you want the student to create a dictionary `x`, that contains three keys: `"a"`, `"b"` and `"c"`. The following solution and sct could be used for this:

~~~
*** =solution
```{python}
x = {'a': 123, 'b':456, 'c':789}
```

*** =sct
```{python}
test_dictionary('x')
```
~~~

- Step 1: if the student submits an empty script, the feedback *Are you sure you defined the dictionary x?* will be presented. You can override this by specifying `undefined_msg` yourself.
- Step 2: if the student submits `x = 123`, the feedback *x is not a dictionary.* will be presented. You can override this message by specifying `not_dictionary_msg` yourself.
- Step 3: if the student submits `x = {'a':123, 'b':456, 'd':78}`, the feedback *Have you specified a key c inside x?* will be presented. You can override this by specifying `key_missing_msg` yourself.
- Step 4: if the student submits `x = {'a':123, 'b':456, 'c':78}`, the feedback *Have you specified the correct value for the key c inside x?* will be presented. You can override this by specifying `incorrect_value_msg` yourself.

**NOTE**: Behind the scenes, `pythonwhat` has to fetch the value of objects from sub-processes. The required 'dilling' and 'undilling' can cause issues for exotic objects. For more information on this and possible errors that can occur, read the *Processes article*.

## test_operator

**THIS FUNCTION IS DEPRECATED AND WILL BE REMOVED IN A FUTURE RELEASE**

**test_operator**(*index=1*, *eq_condition='equal'*, *used=None*, *do_eval=True*, *not_found_msg=None*, *incorrect_op_msg=None*, *incorrect_result_msg=None*, *state=None*)

THIS FUNCTION IS DEPRECATED

Test if operator groups match.

This function compares an operator group in the student's code with the corresponding one in the solution code. It will cause the reporter to fail if the corresponding operators do not match. The fail message that is returned will depend on the sort of fail. We say that one operator group correpsonds to a group of operators that is evaluated to one value (e.g. 3 + 5 * (1/3)).

**Parameters**

- **index** (*int*) – Index of the operator group to be checked. Defaults to 1.
- **eq_condition** (*str*) – how results of operators are compared. Currently, only "equal" is supported, meaning that the result in student and solution process should have exactly the same value.
- **used** (*List[str]*) – A list of operators that have to be in the group. Valid operators are: "+", "-", "*", "/", "%", "**", "<<", ">>", "|", "^", "&" and "//". If the list is None, operators that are in the group in the solution have to be in the student code. Defaults to None.

- **do_eval** (*bool*) – Boolean representing whether the group should be evaluated and compared or not. Defaults to True.
- **not_found_msg** (*str*) – Feedback message if not enough operators groups are found in the student's code.
- **incorrect_op_msg** (*str*) – Feedback message if the wrong operators are used in the student's code.
- **incorrect_result_msg** (*str*) – Feedback message if the operator group evaluates to the wrong result in the student's code.

**Example** Student code:

```
1 + 5 * (3+5)
1 + 1 * 238
```

Solution code:

```
3.1415 + 5
1 + 238
```

SCT:

```
test_operator(index = 2, used = ["+"]) # pass
test_operator(index = 2) # fail
test_operator(index = 1, incorrect_op_msg = "Use the correct operators
↪") # fail
test_operator(index = 1, used = [], incorrect_result_msg = "Incorrect␣
↪result") # fail
```

```
def test_operator(index=1,
                  eq_condition="equal",
                  used=None,
                  do_eval=True,
                  not_found_msg=None,
                  incorrect_op_msg=None,
                  incorrect_result_msg=None)
```

Suppose you want the student to do some very basic operations using the `*` and the `**` operator. You could just ask the student to do some calculations and assign the result to a variable, `result` for example, and check that variable using `test_object()`. However this won't allow you to give the student very tailored feedback. Suppose you want to check if the student uses `**` and tell him/her if he/she doesn't! `test_object()` won't allow you to check this kind of specifics as it only checks resulting objects in both processes. Luckily, you can use another helper function, `test_operator()`.

Say you want the student to calculate the future value of $100 after 6 years. The interest rate 6% and you are using compound interest. This means the result has to be `100 * 1.06 ** 6`, so the solution code would be.

~~~
*** =solution
```{python}
# Calculate the future value of 100 dollar: result
result = 100 * 1.06 ** 6

# Print out the result
print(result)
```
~~~

The SCT might look something like this,

~~~
*** =sct
```{python}
test_operator(index=1)
test_object("result")
test_function("print")
success_msg("Great!")
~~~
```

You can learn about `test_object()` and `test_function()` in the other articles, so those won't be deatiled here. Let's focus on `test_operator()` instead. This function will extract the first operator group from the solution code (`100 * 1.06 ** 6`), run it in the solution process, and compare the result with the result from running the first operator in the student code in the student process. In total, three steps will be tested:

- Did the student define enough operations?
- Does the student use the same operators as the solution?
- Is the result of the operation for the student the same as the one in the solution?

`test_operator()` takes some additional arguments for further customization and tailored feedback messages. For example, you can use it as follows to just check whether the student used the `**` operator in his/her first operation and give custom feedback:

```
*** =sct
```{python}
test_operator(index=1, used=["**"], do_eval=False,
              incorrect_op_msg="A little tip: you should use `**` to do this␣
↪calculation.")
test_object("result")
test_function("print")
success_msg("Great!")
```
```

This SCT will be more forgiving, but the result is still checked with `test_object()` so the student will still have to calculate the correct value. This time, however, it is not checked by `test_operator()` because `do_eval = False`. `used = ["**"]` is used to tell the system to only check on the `**` operator for the first operation group.

**NOTE**: Behind the scenes, `pythonwhat` has to fetch the value of objects from sub-processes. The required 'dilling' and 'undilling' can cause issues for exotic objects. For more information on this and possible errors that can occur, read the *Processes article*.

## test_expression_output

**`test_expression_output`**(*extra_env=None*, *context_vals=None*, *incorrect_msg=None*, *eq_condition='equal'*, *expr_code=None*, *pre_code=None*, *keep_objs_in_env=None*, *state=None*)

Test output of expression.

The code of the student is ran in the active state and the output it generates is compared with the code of the solution. This can be used in nested pythonwhat calls like test_if_else. In these kind of calls, the code of the active state is set to the code in a part of the sub statement (e.g. the body of an if statement). It has various parameters to control the execution of the (sub)expression.

**Parameters**

- **`extra_env`** (*`dict`*) – set variables to the extra environment. They will update the student and solution environment in the active state before the student/solution code in the active
```

state is ran. This argument should contain a dictionary with the keys the names of the variables you want to set, and the values are the values of these variables.

- **context_vals** (*list*) – set variables which are bound in a for loop to certain values. This argument is only useful if you use the function in a test_for_loop. It contains a list with the values of the bound variables.
- **incorrect_msg** (*str*) – feedback message if the output of the expression in the solution doesn't match the one of the student. This feedback message will be expanded if it is used in the context of another test function, like test_if_else.
- **eq_condition** (*str*) – how objects are compared. Currently, only "equal" is supported, meaning that the result in student and solution process should have exactly the same value.
- **expr_code** (*str*) – if this variable is not None, the expression in the student/solution code will not be ran. Instead, the given piece of code will be ran in the student as well as the solution environment and the result will be compared.
- **pre_code** (*str*) – the code in string form that should be executed before the expression is executed. This is the ideal place to set a random seed, for example.
- **keep_obj_in_env** (*list ()*) – a list of variable names that should be hold in the copied environment where the expression is evaluated. All primitive types are copied automatically, other objects have to be passed explicitely.

**Example** Student code:

```
a = 12
if a > 3:
    print('test %d' % a)
```

Soltuion code:

```
a = 4
if a > 3:
    print('test %d' % a)
```

SCT:

```
test_if_else(1,
    body = test_expression_output(
            extra_env = { 'a': 5 },
            incorrect_msg = "Print out the correct things"))
```

This SCT will set a to five, and run the body in both environments (`print('test %d' %a)` in both cases). Since the they print the same output, the test will pass.

```
def test_expression_output(extra_env=None,
                           context_vals=None,
                           incorrect_msg=None,
                           eq_condition="equal",
                           expr_code=None,
                           pre_code=None,
                           keep_objs_in_env=None)
```

`test_expression_output()` is similar to `test_expression_result()`, but instead of checking the result, it checks the output that a single or a set of expressions generates. Typically, this function is used as a sub-test inside other test functions, such as `test_for_loop()` and `test_with()`.

By default, the `test_expression_output()` will execute the 'active expression(s)'; if it's used as a top-level SCT function, that is the entire student submission. If it's used inside the `body` of the `test_for_loop()` function, for example, the entire for loop's body will executed and the output will be compared to the solution. With `expr_code`, you can override this default expression tree. With `pre_code`, you can prepend the execution of the default expression tree with some extra code, for example to set some variables.

Oftentimes, the expression you want to check the output for does not have all variables to its disposal that it requires. Remember that the process in which the expression is evaluated only contains the variables that are available in the global scope. If you're for example running the body of a function definition, this means that the local variables, that are for example passed into the function as arguments, are not all available during execution. To make these variables available, you can set the `extra_env` and `context_vals` arguments. The former is to specify extra variables, with a dictionary. The latter is to specify so called context values; the objects you specify here should not be named; this is to make function definition arguments, iterators in for loops, context variables in `with` constructs, etc, name-independent.

### Example 1

Suppose we want the student to define a function, that loops over the elements in a dictionary, and prints out each key and value, as follows:

~~~
*** =solution
```{python}
def print_dict(my_dict):
    for key, value in my_dict.items():
        print(key + " - " + str(value))
```
~~~

An appropriate SCT for this exercise could be the following (for clarity, we're not using any default messages):

~~~
*** =sct
```{python}
def fun_body_test():
    def for_iter_test():
        example_dict = {'a': 2, 'b': 3}
        test_expression_result(context_vals = [example_dict])
    def for_body_test():
        test_expression_output(context_vals = ['c', 3])
    test_for_loop(for_iter = for_iter_test, body = for_body_test)

test_function_definition('print_dict', body = fun_body_test)
```
~~~

Assuming the student coded the function in the exact same way as the solution, the following things happen:

- `test_function_definition()` is run first: it checks whether `print_dict` is defined, whether the arguments are correctly named and with the correct defaults. Next, it checks the function definition body: it extracts the body of both the student and the solution code, sets the context values for this 'substate', i.e. `"my_dict"`, and then runs `fun_body_test()`.
- Inside `fun_body_test()`, `test_for_loop()` is executed. This function will find the for loop in the function definition body of both student and solution code, and will then run different tests:
  - First, the `for_iter` test is run, which is specified with `for_iter_test()` in this SCT. The `for_iter` part of the `for` loop is extracted, which is `my_dict.items()` in the case of the solution. The context values are still `"my_dict"`. Inside `test_expression_result()`, the context vals are specified, so through `context_vals = [example_dict]`, the variable `my_dict` will now have the value `{'a': 2, 'b':3}` inside the student and solution processes. Next, the currently active

expression (`my_dict.items()`) is executed. The result of calling this expression in both student and solution process is compared.

– Second, the `body` test is run, which is specified iwth `for_body_test()` in this SCT. The `body` part of the `for` loop is extracted, which is `print(key + " - " + str(value))` in the case of the solution. Now, the context values are set to the iterator variables of the `for` loop, so `"key"` and `"value"`. Inside `test_expression_output()`, the context vals are specified: `key` is set to be `'c'`, `value` is set to be `3`. Next, the currently active expression (`print(key + " - " + str(value))`) is executed, and the output it generates is fetched. The output of calling this expression in both student and solution process is compared.

### Example 2

Suppose now that inside the `for` loop of `print_dict()` from the previous example, you each time want to print out the length of the entire dictionary:

~~~
*** =solution
```{python}
def print_dict(my_dict):
    for key, value in my_dict.items():
        print("total length: " + str(len(my_dict)))
        print(key + " - " + str(value))
~~~
```

The SCT from before won't work out of the box, because now you also need a value for `my_dict` inside `test_expression_output()`, the test of the body of the `for` loop, but this value is not available. You cannot specify this value through `context_vals`, because the context variables are already updated to be `"key"` and `"value"`. To be able to test this appropriately, you'll have to set `extra_env` inside `test_expression_output()`:

```
*** =sct
```{python}
def fun_body_test():
    def for_iter_test():
        example_dict = {'a': 2, 'b': 3}
        test_expression_result(context_vals = [example_dict])
    def for_body_test():
        example_dict = {'a': 2, 'b': 3}
        test_expression_output(context_vals = ['c', 3], extra_env = {'my_dict':␣
↪example_dict})

    test_for_loop(for_iter = for_iter_test, body = for_body_test)

test_function_definition('print_dict', body = fun_body_test)
```
```

With this update of the SCT, the exercise will still run fine.

## test_expression_result

**test_expression_result**(*extra_env=None*, *context_vals=None*, *incorrect_msg=None*, *eq_condition='equal'*, *expr_code=None*, *pre_code=None*, *keep_objs_in_env=None*, *error_msg=None*, *state=None*)

Test result of expression.
```

The code of the student is ran in the active state and the result of the evaluation is compared with the result of the solution. This can be used in nested pythonwhat calls like test_if_else. In these kind of calls, the code of the active state is set to the code in a part of the sub statement (e.g. the condition of an if statement). It has various parameters to control the execution of the (sub)expression.

**Parameters**

- **extra_env** (*dict*) – set variables to the extra environment. They will update the student and solution environment in the active state before the student/solution code in the active state is ran. This argument should contain a dictionary with the keys the names of the variables you want to set, and the values are the values of these variables.
- **context_vals** (*list*) – set variables which are bound in a for loop to certain values. This argument is only useful if you use the function in a test_for_loop. It contains a list with the values of the bound variables.
- **incorrect_msg** (*str*) – feedback message if the result of the expression in the solution doesn't match the one of the student. This feedback message will be expanded if it is used in the context of another test function, like test_if_else.
- **eq_condition** (*str*) – how results are compared. Currently, only "equal" is supported, meaning that the result in student and solution process should have exactly the same value.
- **expr_code** (*str*) – if this variable is not None, the expression in the studeont/solution code will not be ran. Instead, the given piece of code will be ran in the student as well as the solution environment and the result will be compared.
- **pre_code** (*str*) – the code in string form that should be executed before the expression is executed. This is the ideal place to set a random seed, for example.
- **keep_obj_in_env** (*list ()*) – a list of variable names that should be hold in the copied environment where the expression is evaluated. All primitive types are copied automatically, other objects have to be passed explicitely.
- **error_msg** (*str*) – Message to override the default error message that is thrown if the expression resulted in an error.

**Example** Student code:

```
a = 12
if a > 3:
    print('test %d' % a)
```

Solution code:

```
a = 4
b = 5
if (a + 1) > (b - 1):
    print('test %d' % a)
```

SCT:

```
test_if_else(1,
    test = test_expression_result(
            extra_env = { 'a': 3 }
            incorrect_msg = "Test if `a` > 3"))
```

This SCT will pass as the condition in the student's code (`a > 3`) will evaluate to the same value as the code in the solution code (`(a + 1) > (b - 1)`), with value of `a` set to `3`.

```
def test_expression_result(extra_env=None,
                           context_vals=None,
                           incorrect_msg=None,
                           eq_condition="equal",
                           expr_code=None,
                           pre_code=None,
                           keep_objs_in_env=None,
                           error_msg=None)
```

`test_expression_result()` works pretty much the same as `test_expression_output()` and takes the same arguments. However, in this case, the expression should be a single expression and can't be a 'tree of expressions', such as the entire body of a function definition for example. Currently, the only places where `test_expression_result()` is used, is inside inherently 'single expression parts' of your code, such as the sequence specification of a `for` loop, the expression of a lambda function, etc.

The example in the `test_expression_output()` article also explains the use of `test_expression_result()`.

## test_object_after_expression

**`test_object_after_expression`**(*name*, *extra_env=None*, *context_vals=None*, *undefined_msg=None*, *incorrect_msg=None*, *eq_condition='equal'*, *expr_code=None*, *pre_code=None*, *keep_objs_in_env=None*, *state=None*)

Test object after expression.

The code of the student is ran in the active state and the the value of the given object is compared with the value of that object in the solution. This can be used in nested pythonwhat calls like test_for_loop. In these kind of calls, the code of the active state is set to the code in a part of the sub statement (e.g. the body of a for loop). It has various parameters to control the execution of the (sub)expression. This test function is ideal to check if a value is updated correctly in the body of a for loop.

**Parameters**

- **name** (*str*) – the name of the object which value has to be checked after evaluation of the expression.
- **extra_env** (*dict*) – set variables to the extra environment. They will update the student and solution environment in the active state before the student/solution code in the active state is ran. This argument should contain a dictionary with the keys the names of the variables you want to set, and the values are the values of these variables.
- **context_vals** (*list*) – set variables which are bound in a for loop to certain values. This argument is only useful if you use the function in a test_for_loop or test_function_definition. It contains a list with the values of the bound variables.
- **incorrect_msg** (*str*) – feedback message if the value of the object in the solution environment doesn't match the one in the student environment. This feedback message will be expanded if it is used in the context of another test function, like test_for_loop.
- **eq_condition** (*str*) – how objects are compared. Currently, only "equal" is supported, meaning that the resulting objects in student and solution process should have exactly the same value.
- **expr_code** (*str*) – if this variable is not None, the expression in the studeont/solution code will not be ran. Instead, the given piece of code will be ran in the student as well as the solution environment and the result will be compared.

- **pre_code** (*str*) – the code in string form that should be executed before the expression is executed. This is the ideal place to set a random seed, for example.
- **keep_obj_in_env** (*list ()*) – a list of variable names that should be hold in the copied environment where the expression is evaluated. All primitive types are copied automatically, other objects have to be passed explicitely.

**Example** Student code:

```
count = 1
for i in range(100):
    count = count + i
```

Solution code:

```
count = 15
for n in range(30):
    count = count + n
```

SCT:

```
test_for_loop(1,
    body = test_object_after_expression("count",
            extra_env = { 'count': 20 },
            contex_vals = [ 10 ])
```

This SCT will pass as the value of *count* is updated identically in the body of the for loop in the student code and solution code.

```
test_object_after_expression(name,
                             extra_env=None,
                             context_vals=None,
                             undefined_msg=None,
                             incorrect_msg=None,
                             eq_condition="equal",
                             pre_code=None,
                             keep_objs_in_env=None)
```

`test_object_after_expression()` is a function that is primarily used to check the correctness of the body of control statements. Through `extra_env` and `context_vals` you can adapt the student/solution environment with manual elements. Next, the 'currently active expression tree', such as the body of a for loop, is executed, and the resulting environment is inspected. This is done for both the student and the solution code, and afterwards the value of the object that you specify in `name` is checked for equality. With pre_code, you can prepend the execution of the default expression tree with some extra code, for example to set some variables.

### Example 1: Function defintion

Suppose you want to student to code up a function `shout()`, that adds three exclamation marks to every word you pass it:

~~~
*** =solution
```{python}
def shout(word):
    shout_word = word + '!!!'
    return shout_word
```
~~~

To test whether the student did this appropriately, you want to first test whether `shout` is a user-defined function, and then whether inside the function, a new variable `shout_word` is created. Finally, you also want to check whether the result of calling `shout('hello')` is correct. The following SCT will do that for us:

~~~
*** =sct
```{python}
test_function_definition('shout',
                         body = test_object_after_expression('shout_word', context_
↪vals = ['anything']),
                         results = [('hello')])
~~~
```

Let's focus on the `body` argument of `test_function_definition()` here, that uses `test_object_after_expression()`. For the other elements, refer to the `test_function_definition()` article.

The first argument of `test_object_after_expression()` tells the system to check the value of `shout_word` after executing the body of the function definition. Which part of the code to execute, the 'expression', is implicitly specified by `pythonwhat`. However, to run correctly, this expression has to know what `word` is. You can specify a value of `word` through the `context_vals` argument. It's a simple list: the first element of the list will be the value for the first argument of the function definition, the second element of the list will be the value for the second argument of the list, and so on. Here, there's only one argument, so a list with a single element, a string (that will be the value of the `word` variable), suffises.

`test_object_after_expression()` will execute the expression, and run it on the solution side and the student side. On the solution side, the value of `shout_word` after the execution will be `'anything!!!'`. If the value on the student code is the same, we can rest assured that `shout_word` has been appropriately defined by the student and the test passes.

### Example 2: for loop

Suppose you want the student to build up a dictionary of word counts based on a list, as follows:

```
*** =solution
```{python}
words = ['it', 'is', 'a', 'the', 'is', 'the', 'a', 'the', 'it']
counts = {}
for word in words:
    if word in counts:
        counts[word] += 1
    else:
        counts[word] = 1
```
```

To check whether the `counts` list was correctly built, you can simply use `test_object()`, but you can also go deeper if it goes wrong. This calls for a `test_correct()` in combination with `test_object()` and `test_for()`, that in its turn uses `test_object_after_expression()`:

```
``` =solution
def check_test():
    test_object('counts')

def diagnose_test():
    body_test():
        test_object_after_expression('counts',
                                     extra_env = {'counts': {'it': 1}},
                                     context_vals = ['it'])
```

```
        test_object_after_expression('counts',
                                     extra_env = {'counts': {'it': 1}},
                                     context_vals = ['is'])
    test_for_loop(index = 1,
                  test = test_expression_result(), # Check if correct iterable used
                  body = body_test)

test_correct(check_test, diagnose_test)
```
```

Let's focus on the `body_test` for the for loop. Here, we're using `test_object_after_expression()` twice.

In the first function call, we override the environment so that `counts` is a dictionary with a single key and value. Also, the context value, `word` in this case (the iterator of the `for` loop), is set to `it`. In this case, the body of the for loop - making abstraction of the if-else test - should increment the value of the value, without adding a new key.

In the second function call, we override the environment so that `counts` is again a dictionary with a single key and value. This time, the context value is set to `is`, so a value that is not yet in the `counts` dictionary, so this should lead to a `counts` dictionary with two elements.

As in the first example, `test_object_after_expression()` sets the environment variables and context values, runs the expression (in this case the entire body of the `for` loop), and then inspects the value of `counts` after this expression. The combination of the two `test_object_after_expression()` calls here, will indirectly check whether both the if and else part of the body has been correctly implemented.

## test_object_accessed

**`test_object_accessed`**(*name*, *times=1*, *not_accessed_msg=None*, *state=None*)

Test if object accessed

Checks whether an object, or the attribute of an object, are accessed

**Parameters**

- **`name`** (*`str`*) – the name of the object that should be accessed; can contain dots (for attributes)
- **`times`** (*`int`*) – how often the object specified in name should be accessed.
- **`not_accessed_msg`** (*`str`*) – custom feedback message when the object was not accessed.

### Examples

Student code

```
import numpy as np
arr = np.array([1, 2, 3])
x = arr.shape
```

Solution code
```

```
import numpy as np
arr = np.array([1, 2, 3])
x = arr.shape
t = arr.dtype
```

SCT

`test_object_accessed("arr")`: pass.
`test_object_accessed("arr.shape")`: pass.
`test_object_accessed("arr.dtype")`: fail.

```
def test_object_accessed(name,
                         times=1,
                         not_accessed_msg=None)
```

With `test_object()`, you can check whether a student correctly created an object. However, in some cases, you might also be interested whether the student actually used this object to for example assign another object. `test_object_accessed()` makes this possible; it is also possible to test object attributes.

The `name` argument should be a string that specifies the name of the object, or the attribute of a certain object, for which you want to check if it was accesses. If the object resides inside a package, such as `pi` in the `math` package, use `"math.pi"`. With `times`, you can specify how often the object or attribute should have been accessed. With `not_accessed_msg` you can override the automatically generated feedback message in case `name` hasn't been accessed often enough according to `times`.

### Example

To show how everything works, suppose you have the following submission of a student:

~~~
```
import numpy as np
import math as m
arr = np.array([1, 2, 3])
x = arr.shape
print(arr.data)
print(m.e)
```
~~~

Let's have a look at some SCT function calls that either pass or fail and why.

- `test_object_accessed("arr")` - PASS: The object `arr` is accessed twice (in `arr.shape` and `arr.data`)
- `test_object_accessed("arr", times=3)` - FAIL: The objet `arr` is only accessed twice.
- `test_object_accessed("arr.shape")` - PASS: The `shape` attribute of `arr` is accessed once.
- `test_object_accessed("math.e")` - PASS: The object `e` inside the `math` package is accessed once (the student uses the alias `m`, but that is not a problem. In case of an error, the automatically generated message will take this into account.)

## parts_cheatsheet

### test_list_comp

```
[BODY for i in COMP_ITER if IFS[0] if IFS[1]]
```

### test_dict_comp

```
{ KEY : VALUE for k, v in COMP_ITER if IFS[0] if IFS[1] }
```

### test_generator_exp

```
(BODY for i in COMP_ITER if IFS[0] if IFS[1])
```

### test_for_loop

```
for i in FOR_ITER:
    BODY
else:
    ORELSE
```

*yes, you can put an else statement at the end!*

### test_if_else

```
if TEST:
    BODY
else:
    ORELSE
```

or, in the case of elif statements...

```
if TEST:
    BODY
ORELSE
```

### test_lambda

```
lambda x: BODY
```

### test_try_except

```
try:
    BODY
except BaseException:
    HANDLERS['BaseException']
```

```
except:
    HANDLERS['all']
else:
    ORELSE
finally:
    FINALBODY
```

### test_while

```
while TEST:
    BODY
else:
    ORELSE
```

### test_with

```
with CONTEXT_TEST as context_var:
    BODY
```

### test_function_definition

```
def f(a, b):
    BODY
```

## test comprehensions

```
def test_list_comp(index=1,
                   not_called_msg=None,
                   comp_iter=None,
                   iter_vars_names=False,
                   incorrect_iter_vars_msg=None,
                   body=None,
                   ifs=None,
                   insufficient_ifs_msg=None,
                   expand_message=True)

def test_generator_exp(index=1,
                       not_called_msg=None,
                       comp_iter=None,
                       iter_vars_names=False,
                       incorrect_iter_vars_msg=None,
                       body=None,
                       ifs=None,
                       insufficient_ifs_msg=None,
                       expand_message=True)

def test_dict_comp(index=1,
                   not_called_msg=None,
                   comp_iter=None,
                   iter_vars_names=False,
                   incorrect_iter_vars_msg=None,
                   key=None,
                   value=None,
                   ifs=None,
                   insufficient_ifs_msg=None,
                   expand_message=True)
```

Currently, functionality to test list comprehensions, generator expressions and dictionary comprehensions is implemented. If you look at the signatures, you'll see that the arguments for `test_list_comp()` and `test_generator_exp()` are identical. Syntactically, there is close to no difference between list comprehensions and generator expressions, so all tests and settings apply for both cases. For `test_dict_comp()` there's only a small difference: the arguments `key` and `value` instead of the `body` argument, so that you can test the `key` part of the dictionary comprehension seperately from the `value` comprehension.

The above functions work pretty similarly to `test_for_loop()`, with some additions and customizations here and there. Let's go over the argments:

- `index`: the number of the comprehension in the submission to test. (this is specific to each comprehension, if there's one list and one dict comprehension you need `index=1` twice.)
- `not_called_msg`: Custom message in case the comprehension was not coded (or there weren't enough comprehensions).
- `comp_iter`: sub SCT to check the sequence part of the comprehension. Specify this through another function definition or a lambda function.
- `iter_vars_names`: whether or not the iterator variables should match the ones in the solution.
- `incorrect_iter_vars_msg`: Custom message in case the iterator variables don't match the solution (if `iter_vars_names` is `True`) or if the number of iterator variables doesn't correspond to the solution.
- `body`, `key`, `value`: sub SCTs to check the body part (for list comps and generator expressions) or the key and value part of a dictionary comprehension.
- `ifs`: list of sub-SCTs to check each of the ifs specified inside the comprehension. If you specify `ifs`, make sure that the number of sub-SCTs corresponds exactly to the number of ifs that are in the solution.
- `insufficient_ifs`: custom message in case the student coded less ifs than the corresponding comp in the solution.
- `expand_message`: whether or not to expand feedback messages from sub-SCTs with more information about where in the list comprehension they occur.

### Example 1: List comprehension

Suppose you want the student to code a list comprehension like below:

~~~
*** =solution
```{python}
x = {'a': 2, 'b':3, 'c':4, 'd':'test'}
[key + str(val) for key,val in x.items() if isinstance(key, str) if isinstance(val,␣
↪int)]
~~~
```

The following SCT will test several parts of this list comprehension, and relies on automatic feedback messages everywhere:

```
*** =sct
```{python}
```

```
test_list_comp(index=1,
           comp_iter=lambda: test_expression_result(),
           iter_vars_names=True,
           body=lambda: test_expression_result(context_vals = ['a', 2]),
           ifs=[lambda: test_function_v2('isinstance', params = ['obj'], do_eval =␣
↪[False]),
                lambda: test_function_v2('isinstance', params = ['obj'], do_eval =␣
↪[False])])
```

By setting `iter_vars_names` to `True`, `pythonwhat` will check that the student actually used the iterator variables `key` and `val`. Notice that in the sub SCT for the body, `context_vals` are used to set the `key` and `val` iterator variables before the expression is tested. This is similar to how things work in `test_for_loop()`. Notice also that inside the list of if sub-SCTs, `do_eval` is false, because the values `key` and `val` are not available there (setting context vals is currently only possible inside `test_expression_*()` functions).

`test_list_comp()` will generate a bunch of meaningful automated messages depending on which error the student made:

```
submission: <empty>
feedback: "The system wants to check the first list comprehension you defined but hasn
↪'t found it."

submission: [key for key in x.keys()]
feedback: "Check your code in the iterable part of the first list comprehension.␣
↪Unexpected expression: expected `dict_items([('a', 2), ('b', 3), ('c', 4), ('d',
↪'test')])`, got `dict_keys(['a', 'b', 'c', 'd'])` with values."

submission: [a + str(b) for a,b in x.items()]
feedback: "Have you used the correct iterator variables in the first list␣
↪comprehension? Make sure you use the correct names!"

submission: [key + '_' + str(val) for key,val in x.items()]
feedback: "Check your code in the body of the first list comprehension. Unexpected␣
↪expression: expected `a2`, got `a_2` with values."

submission: [key + str(val) for key,val in x.items()]
feedback: "Have you used 2 ifs inside the first list comprehension?"

submission: [key + str(val) for key,val in x.items() if hasattr(key, 'test') if␣
↪hasattr(key, 'test')]
feedback: "Check your code in the first if of the first list comprehension. Have you␣
↪called `isinstance()`?"

submission: [key + str(val) for key,val in x.items() if isinstance(key, str) if␣
↪hasattr(key, 'test')]
feedback: "Check your code in the second if of the first list comprehension. Have you␣
↪called `isinstance()`?"

submission: [key + str(val) for key,val in x.items() if isinstance(key, str) if␣
↪isinstance(key, str)]
feedback: "Check your code in the second if of the first list comprehension. Did you␣
↪call `isinstance()` with the correct arguments?"

submission: [key + str(val) for key,val in x.items() if isinstance(key, str) if␣
↪isinstance(val, str)]
feedback: "Great work!"
```

NOTE: the "check your code in the ... of the first list comprehension" parts are included because `expand_message = True`.

You can also update SCT to override all automatically generated messages, either inside `test_list_comp()` itself or inside the sub-SCTs:

~~~
*** =sct
```{python}
test_list_comp(index=1,
           not_called_msg='notcalled',
           comp_iter=lambda: test_expression_result(incorrect_msg = 'iterincorrect'),
           iter_vars_names=True,
           incorrect_iter_vars_msg='incorrectitervars',
           body=lambda: test_expression_result(context_vals = ['a', 2], incorrect_msg␣
↪= 'bodyincorrect'),
           ifs=[lambda: test_function_v2('isinstance', params = ['obj'], do_eval =␣
↪[False], not_called_msg = 'notcalled1', incorrect_msg = 'incorrect2'),
                lambda: test_function_v2('isinstance', params = ['obj'], do_eval =␣
↪[False], not_called_msg = 'notcalled2', incorrect_msg = 'incorrect2')],
           insufficient_ifs_msg='insufficientifs')
```
~~~

In this case, you get the following feedback for different submissions:

```
submission:
feedback: "notcalled"

submission: [key for key in x.keys()]
feedback: "Check your code in the iterable part of the first list comprehension.␣
↪iterincorrect"

submission: [a + str(b) for a,b in x.items()]
feedback: "incorrectitervars"

submission: [key + '_' + str(val) for key,val in x.items()]
feedback: "Check your code in the body of the first list comprehension. bodyincorrect"

submission: [key + str(val) for key,val in x.items()]
feedback: "insufficientifs"

submission: [key + str(val) for key,val in x.items() if hasattr(key, 'test') if␣
↪hasattr(key, 'test')]
feedback: "Check your code in the first if of the first list comprehension. notcalled1
↪"

submission: [key + str(val) for key,val in x.items() if isinstance(key, str) if␣
↪hasattr(key, 'test')]
feedback: "Check your code in the second if of the first list comprehension.␣
↪notcalled2"

submission: [key + str(val) for key,val in x.items() if isinstance(key, str) if␣
↪isinstance(key, str)]
feedback: "Check your code in the second if of the first list comprehension.␣
↪incorrect2"

submission: [key + str(val) for key,val in x.items() if isinstance(key, str) if␣
↪isinstance(val, str)]
feedback: "Great work!"
```

### Example 2: Generator Expressions

An example here won't be necessary, because it works the exact same way as in Example 1, with the only difference that in automated feedback, "list comprehension" is replaced with "generator expression".

### Example 3: Dictionary Comprehensions

Suppose you want the student to code a dictionary comprehension like below:

~~~
*** =solution
```{python}
x = {'a': 2, 'b':3, 'c':4, 'd':'test'}
[key + str(val) for key,val in x.items() if isinstance(key, str) if isinstance(val,␣
↪int)]
~~~
```

The following SCT will test several parts of this list comprehension, and relies on automatic feedback messages everywhere:

```
*** =sct
```{python}
test_list_comp(index=1,
           comp_iter=lambda: test_expression_result(),
           iter_vars_names=True,
           body=lambda: test_expression_result(context_vals = ['a', 2]),
           ifs=[lambda: test_function_v2('isinstance', params = ['obj'], do_eval =␣
↪[False]),
                lambda: test_function_v2('isinstance', params = ['obj'], do_eval =␣
↪[False])])
```
```

Again, customized messages are generated for different cases:

```
submission:
feedback: "The system wants to check the first dictionary comprehension you defined␣
↪but hasn't found it."

submission: { a:a for a in lst[1:2] }
feedback: "Check your code in the iterable part of the first dictionary comprehension.
↪ Unexpected expression: expected `['this', 'is', 'a', 'list']`, got `['is']` with␣
↪values."

submission: { a:a for a in lst }
feedback: "Have you used the correct iterator variables in the first dictionary␣
↪comprehension? Make sure you use the correct names!"

submission: { el + 'a':str(el) for el in lst }
feedback: "Check your code in the key part of the first dictionary comprehension.␣
↪Unexpected expression: expected `a`, got `aa` with values."

submission: { el:str(el) for el in lst }
feedback: "Check your code in the value part of the first dictionary comprehension.␣
↪Unexpected expression: expected `1`, got `a` with values."

submission: { el:len(el) for el in lst }
feedback: "Have you used 1 ifs inside the first dictionary comprehension?"
```
```

```
submission: { el:len(el) for el in lst if isinstance('a', str)}
feedback: "Check your code in the first if of the first dictionary comprehension. Did␣
↪you call `isinstance()` with the correct arguments?"

submission: { el:len(el) for el in lst if isinstance(el, str)}
feedback: "Great work!"
```

## test_for_loop

**test_for_loop**(*index=1*, *for_iter=None*, *body=None*, *orelse=None*, *expand_message=True*, *state=None*)
Test parts of the for loop.

This test function will allow you to extract parts of a specific for loop and perform a set of tests specifically on these parts. A for loop consists of two parts: the sequence, *for_iter*, which is the values over which are looped, and the *body*. A for loop can have a else part as well, *orelse*, but this is almost never used.:

```
for i in range(10):
    print(i)
```

Has `range(10)` as the sequence and `print(i)` as the body.

**Parameters**

- **index** (*int*) – index of the function call to be checked. Defaults to 1.
- **for_iter** – this argument holds the part of code that will be ran to check the sequence of the for loop. It should be passed as a lambda expression or a function. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the sequence part of the for loop.
- **body** – this argument holds the part of code that will be ran to check the body of the for loop. It should be passed as a lambda expression or a function. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the body of the for loop.
- **orelse** – this argument holds the part of code that will be ran to check the else part of the for loop. It should be passed as a lambda expression or a function. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the else part of the for loop.
- **expand_message** (*bool*) – if true, feedback messages will be expanded with `in the ___ of the for loop on line ___`. Defaults to True. If False, `test_for_loop()` will generate no extra feedback.

**Example** Student code:

```
for i in range(10):
    print(i)
```

Solution code:

```
for n in range(10):
    print(n)
```

SCT:

```
test_for_loop(1,
    for_iter = test_function("range"),
    body = test_expression_output(context_val = [5])
```

This SCT will evaluate to True as the function `range` is used in the sequence and the function `test_exression_output()` will pass on the body code.

```
test_for_loop(index=1,
              for_iter=None,
              body=None,
              orelse=None,
              expand_message=True)
```

As the name suggesets, you can use `test_for_loop()` to test if a for loop was properly coded. Similar to how `test_if_else()` and `test_while_loop()` works, `test_for_loop()` parses the for loop in the student's submission and breaks it up into its composing parts. Next, it also parses the for loop in the solution solution, and compares the parts between student submission and solution. It does this through sub-SCTs that you specify in `cond_test` and `expr_test`.

### Example 1

Suppose you want the student to implement an algorithm that calculates fibonacci's row (until `n = 20`) using a simple for loop. The solution could look like this:

~~~
*** =solution
```{python}
# Initialise the row
fib = [0, 1]

# Update the row correctly each loop
for n in range(2, 20):
    fib.append(fib[n-2] + fib[n-1])
```
~~~

An SCT to accompany this exercise could be the following:

~~~
*** =sct
```{python}
def test_for_iter():
    "You have to iterate over `range(2, 20)`"
    test_function("range",
                  not_called_msg=msg,
                  incorrect_msg=msg)

def test_for_body():
    msg = "Make sure your row, `fib`, updates correctly"
    test_object_after_expression("fib",
                                 extra_env={ "fib": [0, 1, 1, 2] },
                                 context_vals=[4],
                                 undefined_msg=msg,
                                 incorrect_msg=msg)
test_for_loop(index=1,
              for_iter=test_for_iter,
              body=test_for_body)
```
~~~

Notice that two self-defined functions, `test_for_iter()` and `test_for_body()` are used to specify the sub-SCTs for the different parts in the `for` loop. With `index = 1`, you tell `pythonwhat` that you want to check the first `for` loop you find in the student submission with the first `for` loop in the solution.

The `for_iter` part of `test_for_loop()` tests whether the loop with index 1 loops over the correct range. The tests in this sub-SCT are run on the sequence part of the loop, which in this case for the solution is `range(2, 20)`. With `test_function()`, we can test this. In other cases, you could use e.g. `test_expression_result()`, to test the result of the sequence part.

The `body` part of `test_for_loop()` tests whether `fib` is updated correctly. The tests in this sub-SCT are run on the body of the loop. The `test_object_after_expression()`. This function will test an object after the active expression is run in the student and solution process. In this case it will check if `fib` is updated the same in the student and solution process after one loop through the body of the `for`. Two important arguments for `test_object_after_expression()` are:

- `extra_env = { "fib": [0, 1, 1, 2] }`: when running the body of the for loop, the process will be updated with these extra environment variables. In this case this means that before the body is ran, `fib` will be initialised to `[0, 1, 1, 2]`.
- `context_vals = [4]`: this argument contains the values of the loop's variable. In the solution code, for example, there will be one: `n`. This means that `n` will be initialised to `4` in the solution process when the body of the for loop is run. The student can give any name to `n`, as long as the functionality remains the same.

You may have noticed that the helper functions that are used within `test_for_loop()` contain feedback messages as well. When they are used within a `test_for_loop()`, these messages will automatically be extended with "in the ___ of the for loop on line ___.". To avoid this extension, you could set the option `expand_message = False` in `test_for_loop()`.

Example 2: Multiple context vals

If you have multiple context vals, things largely work the same way. Suppose you want somebody to print out the keys and values of a dictionary as follows:

~~~
*** =solution
```{python}
my_dict = {'a': 1, 'b': 2, 'c': 3}
for k, v in my_dict.items():
    print(k + ' - ' + str(v))
~~~
```

An appropriate SCT would be:

```
*** =sct
```{python}
test_object('my_dict')
test_for_loop(index=1,
              for_iter = lambda: test_expression_result(),
              body = lambda: test_expression_output(context_vals = ['a', 1]))
```
```

In this case, when you're checking the output of the body of the `for` loop, you're telling `k` to be `'a'` and `v` to be `1`.

## test_if_else

**test_if_else**(*index=1*, *test=None*, *body=None*, *orelse=None*, *expand_message=True*, *use_if_exp=False*, *state=None*)

Test parts of the if statement.
```

This test function will allow you to extract parts of a specific if statement and perform a set of tests specifically on these parts. A for loop consists of three potential parts: the condition test, `test`, which specifies the condition of the if statement, the `body`, which is what's executed if the condition is True and a else part, `orelse`, which will be executed if the condition is not True.:

```
if 5 == 3:
    print("success")
else:
    print("fail")
```

Has `5 == 3` as the condition test, `print("success")` as the body and `print("fail")` as the else part.

**Parameters**

- **`index`** (*int*) – index of the function call to be checked. Defaults to 1.
- **`test`** – this argument holds the part of code that will be ran to check the condition test of the if statement. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the condition test of the if statement.
- **`body`** – this argument holds the part of code that will be ran to check the body of the if statement. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the body of the if statement.
- **`orelse`** – this argument holds the part of code that will be ran to check the else part of the if statement. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the else part of the if statement.
- **`expand_message`** (*bool*) – if true, feedback messages will be expanded with `in the ____ of the if statement on line ____`. Defaults to True. If False, `test_if_else()` will generate no extra feedback.

**Example** Student code:

```
a = 12
if a > 3:
    print('test %d' % a)
```

Solution code:

```
a = 4
if a > 3:
    print('test %d' % a)
```

SCT:

```
test_if_else(1,
    body = test_expression_output(
            extra_env = { 'a': 5 }
            incorrect_msg = "Print out the correct things"))
```

This SCT will pass as `test_expression_output()` is ran on the body of the if statement and it will output the same thing in the solution as in the student code.

```
test_if_else(index=1,
             test=None,
             body=None,
             orelse=None,
             expand_message=True)
```

`test_if_else()` allows you to robustly check `if` statements, optionally extended with `elif` and `else` components. For each of the components of an if-else construct `test_if_else()` takes several 'sub-SCTs'. These 'sub-SCTs', that you have to pass in the form of lambda functions or through a function that defines all tests, are executed on these separate parts of the submission.

### Example 1

Suppose an exercise asks the student to code up the following if-else construct:

~~~
*** =solution
```{python}
# a is set to 5
a = 5

# If a < 5, print out "It's small", else print out "It's big"
if a < 5:

else:
  print("It's big")
~~~
```

The `if-else` construct here consists of three parts:

- The condition to check: `a < 5`. The `test` argument of `test_if_else()` specifies the sub-SCT to test this.
- The body of the `if` statement: `print("It's small")`. The `body` argument of `test_if_else()` specifies the sub-SCT to test this.
- The else part: `print("It's big")`. The `orelse` argument of `test_if_else()` specifies the sub-SCT to ttest this.

You can thus write our SCT as follows. Notice that for the `test` argument a function is used to specify different tests; for the `body` and `orelse` arguments two lambda functions suffise.

```
*** =sct
```{python}
def sct_on_condition_test():
  test_expression_result({"a": 4})
  test_expression_result({"a": 5})
  test_expression_result({"a": 6})

test_if_else(index = 1,
             test = sct_on_condition_test,
             body = lambda: test_function("print")
             orelse = lambda: test_function("print"))
```
```

### The `test` part

Have a look at the `sct_on_condition_test()`, that is used to specify the sub-SCT for the `test` part of the if-else-construct, so `a < 5`. It contains three calls of the `test_expression_result` function. These functions are executed in a 'narrow scope' that only considers the condition of the student code, and the condition of the solution code.
```

More specifically, `test_expression_result({"a": 5})` will check whether executing the `if` condition that the student coded when `a` equals 5 leads to the same result as executing the `if` condition that is coded in the solution when `a` equals 5. That way, you can robustly check the validity of the `if` test. There are three `test_expression_result()` calls to see if the condition makes sense for different inputs.

Suppose that the student incorrectly used the condition `a < 6` instead of `a < 5`. `test_expression_result({"a": 5})` will see what the result is of `a < 6` if `a` equals 5. The result is `True`. Next, it checks the result of `a < 5`, the `if` condition of the solution, which is `False`. There is a mismatch between the 'student result' and the 'solution result', and a meaningful feedback messages is generated.

### The `body` and `orelse` parts

In a similar way, the functions that are used as lambda functions in both the `body` and `orelse` part, will also be executed in a 'narrow scope', where only the `body` and `orelse` part of the student's submission and the solution are used.

## test_if_exp

**`test_if_exp`**(*index=1*, *test=None*, *body=None*, *orelse=None*, *expand_message=True*, *state=None*)

Test parts of the if statement.

This test function will allow you to extract parts of a specific if statement and perform a set of tests specifically on these parts. A for loop consists of three potential parts: the condition test, `test`, which specifies the condition of the if statement, the `body`, which is what's executed if the condition is True and a else part, `orelse`, which will be executed if the condition is not True.:

```
if 5 == 3:
    print("success")
else:
    print("fail")
```

Has `5 == 3` as the condition test, `print("success")` as the body and `print("fail")` as the else part.

**Parameters**

- **index** (*int*) – index of the function call to be checked. Defaults to 1.
- **test** – this argument holds the part of code that will be ran to check the condition test of the if statement. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the condition test of the if statement.
- **body** – this argument holds the part of code that will be ran to check the body of the if statement. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the body of the if statement.
- **orelse** – this argument holds the part of code that will be ran to check the else part of the if statement. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the else part of the if statement.
- **expand_message** (*bool*) – if true, feedback messages will be expanded with `in the ___ of the if statement on line ___`. Defaults to True. If False, `test_if_else()` will generate no extra feedback.

**Example** Student code:

```
a = 12
if a > 3:
    print('test %d' % a)
```

Solution code:

```
a = 4
if a > 3:
    print('test %d' % a)
```

SCT:

```
test_if_else(1,
    body = test_expression_output(
            extra_env = { 'a': 5 }
            incorrect_msg = "Print out the correct things"))
```

This SCT will pass as `test_expression_output()` is ran on the body of the if statement and it will output the same thing in the solution as in the student code.

`test_if_exp` is a wrapper around `test_if_else`, which tells it to look for inline `if` expressions. As such, it uses the same arguments. *See*.

### What is an inline `if` expression?

An inline `if` expression looks like..

```
x = 'a' if True else 'b'
```

This is in contrast to an `if` block, which looks like..

```
if True:
    x = 'a'
else:
    x = 'b'
```

### Parts

This test tries to break code into 3 parts, BODY, TEST, and ORELSE. The table below shows an example inline `if` expression on the left, and the parts that would be extracted on the right.

| code | parts breakdown | | ——————————- | ————————– | | `x = 'a' if True else 'b'` | `x = BODY if TEST else ORELSE` |

### Nested `if` expressions

Just like `test_if_else`, `test_if_exp` will not find a nested `if` expression. Instead, the nested portion will be inside one of the parts. For example, below is an exercise with an `if` expression in the ORELSE part of another `if` expression.

*** =solution

```
x = 'a' if True else ('b' if False else 'c')
```

*** =sct

```
test_if_exp(orelse=lambda: test_if_exp(orelse=lambda: test_student_typed('c')))
```

The SCT above tests that the student typed 'c' in the ORELSE part of the inner `if` expression. In parts, this looks like..

```
BODY1 if TEST1 else (ORELSE1 = BODY2 if TEST2 else ORELSE2)
```

## test_try_except

**test_try_except**(*index=1*, *not_called_msg=None*, *body=None*, *handlers={}*, *except_missing_msg=None*, *orelse=None*, *orelse_missing_msg=None*, *finalbody=None*, *finalbody_missing_msg=None*, *expand_message=True*, *state=None*)

Test whether the student correctly coded a *try-except* block.

This function allows you to test specific parts of a try-except block. A try-except block consists of 4 parts: a body, error handlers, plus (rarely) an else and final block.

```
try:                print(hello)
except NameError: print('hello what?')
except:             print('unexplained error')
else:               print('else block')
finally:            print('final block')
```

**Parameters**

- **index** (*int*) – index of the try-except block to check.
- **not_called_msg** – override the default message when too few try-except blocks found in student code.
- **body** – sub-sct to test the code of the *try* block.
- **handlers** – a dictionary, where the keys are the error classes you expect the student to capture (for the general *except:*, use *'all'*), and the values are sub-SCTs for each of these *except* blocks.
- **except_missing_message** – override the default message when a expect block in the handlers arg is missing.
- **orelse** – similar to body, but for the else block.
- **finalbody** – similar to body, but for the finally block.
- **_missing_msg** – custom messages if the orelse, or finalbody pieces are missing.

```
def test_try_except(index=1,
                    not_called_msg=None,
                    body=None,
                    handlers={},
                    except_missing_msg = None,
                    orelse=None,
                    orelse_missing_msg=None,
                    finalbody=None,
                    finalbody_missing_msg=None,
                    expand_message=True)
```

With `test_try_except`, you can check whether the student correctly coded a `try-except` block.

As usual, `index` controls which try-except block to check. With `not_called_msg` you can choose a custom message to override the automatically defined message in case not enough try-except blocks weren't found in the student code. `body` is a sub-sct to test the code of the `try` block. `orelse` and `finalbody` work the same way, but here there are also `_msg` arguments to provide custom messages in case these parts ar missing. Finally, there's also `handlers` and `except_missing_msg`. `handlers` should be a dictionary, where the keys are the error classes you expect the student to capture (for the general `except:`, use `'all'`), and the values are sub-SCTs for each of these `except` blocks. An `except` block is only checked for existence and correctness if you mention it inside `handlers`. If it is not available, an automatic message will be generated, but this can ge overriden with `expect_missing_msg`.

Note: For more information on sub-SCTs, visit *part checks*.

### Example 1

Suppose you want to student to code up the following (completely useless) piece of Python code:

~~~
*** =solution
```{python}
try:
    x = max([1, 2, 'a'])
except TypeError as e:
    x = 'typeerror'
except ValueError:
    x = 'valueerror'
except (ZeroDivisionError, IOError) as e:
    x = e
except :
    x = 'someerror'
else :
    passed = True
finally:
    print('done')
~~~
```

To test each and every part of this model solution, you can use the following SCT:

```
*** =sct
```{python}
import collections
handlers = collections.OrderedDict()
handlers['TypeError'] = lambda: test_object_after_expression('x')
handlers['ValueError'] = lambda: test_object_after_expression('x')
handlers['ZeroDivisionError'] = lambda: test_object_after_expression('x', context_
↪vals = ['anerror'])
handlers['IOError'] = lambda: test_object_after_expression('x', context_vals = [
↪'anerror'])
handlers['all'] = lambda: test_object_after_expression('x')
test_try_except(index = 1,
                body = lambda: test_function("max"),
                handlers = handlers,
                orelse = lambda: test_object_after_expression('passed'),
                finalbody = lambda: test_function('print'))
```
```

Notice that:
```

- We use the `OrderedDict()` from the `collections` module so that the dictionary we pass in the `handlers` argument is always gone through in the same order.
- We can use `context_vals` to initalize the context value, `e` in this case.

## test_while_loop

**`test_while_loop`**(*index=1*, *test=None*, *body=None*, *orelse=None*, *expand_message=True*, *state=None*)

Test parts of the while loop.

This test function will allow you to extract parts of a specific while loop and perform a set of tests specifically on these parts. A while loop generally consists of two parts: the condition test, `test`, which is the condition that is tested each loop, and the `body`. A for while can have a else part as well, `orelse`, but this is almost never used.:

```
a = 10
while a < 5:
    print(a)
    a -= 1
```

Has `a < 5` as the condition test and *print(i)* as the body.

**Parameters**

- **index** (*int*) – index of the function call to be checked. Defaults to 1.
- **test** – this argument holds the part of code that will be ran to check the condition test of the while loop. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the condition test of the while loop.
- **body** – this argument holds the part of code that will be ran to check the body of the while loop. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the body of the while loop.
- **orelse** – this argument holds the part of code that will be ran to check the else part of the while loop. It should be passed as a lambda expression or a function definition. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the else part of the while loop.
- **expand_message** (*bool*) – if true, feedback messages will be expanded with `in the ___ of the while loop on line ___`. Defaults to True. If False, *test_for_loop()* will generate no extra feedback.

**Example**

Student code:

```
a = 10
while a < 5:
    print(a)
    a -= 1
```

Solution code:

```
a = 20
while a < 5:
    print(a)
    a -= 1
```

```
```

SCT:

```
test_while_loop(1,
        test = test_expression_result({"a": 5}),
        body = test_expression_output({"a": 5}))
```

This SCT will evaluate to True as condition test will have thes same result in student and solution code and *test_exression_output()* will pass on the body code.

```
test_while_loop(index=1,
                test=None,
                body=None,
                orelse=None,
                expand_message=True)
```

Since a lot of the logic of `test_if_else()` and `test_for_loop()` can be applied to `test_while_loop()`, this article is limited to an example. For more info see the wiki on `test_if_else()` and `test_for_loop()`, or the documentation of `test_while_loop()`.

~~~
*** =solution
```{python}
a = 10
while a > 5:
  print("%s is bigger than 5" % a)
  a -= 1
~~~

*** =sct
```{python}
def sct_on_condition_test():
  test_expression_result({"a": 4})
  test_expression_result({"a": 5})
  test_expression_result({"a": 6})

test_while_loop(index = 1,
                test = sct_on_condition_test,
                body = lambda: test_expression_output({"a":4}))
```
```

## test_with

**`test_with`**(*index*, *context_vals=False*, *context_tests=None*, *body=None*, *undefined_msg=None*, *context_vals_len_msg=None*, *context_vals_msg=None*, *expand_message=True*, *state=None*)

Test a with statement. with open_file('...') as bla:

[ open_file('...').__enter__() ]

**with open_file('...') as file:** [ ]

```
def test_with(index,
              context_vals=False,
              context_tests=None,
              body=None,
              undefined_msg=None,
```
```

```
              context_vals_len_msg=None,
              context_vals_msg=None,
              expand_message=True)
```

In Python, one can build so-called context managers with the `with` statement.

Have a look at an example of such a context manager:

```python
with open('something.txt') as file1, open('something_else.csv') as file2: # the␣
↪contexts
    # body of the with statement
    # do something with file1 and file2
    # ...
```

Two important parts can be distinguished: the contexts that are being opened and the body, in which operations are done with these contexts. In this example, two contexts are defined: `open('something.txt')` and `open('something_else.csv')`. The context can be given names (this is optional). In the example, the first one will be called `file1` after the `with` statement, and the second one `file2`.

`test_with()` is written to allow you to test all these parts of the `with` statement separately.

## Example 1

Suppose you want the student to code something as follows:

~~~
*** =solution
```{python}
with open('moby_dick.txt') as moby, open('lotr.txt') as lotr:
    print("First line of Moby Dick: %r." % moby.readline())
    print("First line of The Lord of The Rings: The Two Towers: %r." % lotr.
↪readline())
~~~

In this case you want to test two things: you want the student to open up the correct context and you want them to print out the correct information. Let's assume that how the context are named is not important to you. The solution uses `moby` and `lotr`, but the student can use any name he or she wants. Note these names will not be tested by default, but you can change that by setting `context_vals = True`.

In the SCT, we specify a sub-SCT for `context_tests` and for `body`. The former tests the contexts, the latter tests the body. As before, you can specify these sub-SCTs through lambda functions or a separate function definition:

~~~
*** =sct
```{python}
def test_with_body():
    test_function('print', 1)
    test_function('print', 2)

test_with(1,
          context_tests = [
              lambda: test_function('open'),
              lambda: test_function('open')
         ],
         body = test_with_body)
```
~~~

Different from before, htough, `context_tests` expects a list of lambda functions or customly defined functions. The index in this list of functions represents the context against which the SCTs will be tested. The first lambda/custom function in `context_tests` will be tested against the first context. The second lambda/custom function in `context_tests` will be tested against the second context. If only one function is given in `context_tests`, only the first context will be tested. The `body` argument requires one lambda/custom function to be passed, this contains the sub-SCT that is run against the `with` statements' body.

## test_function_definition

**`test_function_definition`**(*name*, *arg_names=True*, *arg_defaults=True*, *body=None*, *results=None*, *outputs=None*, *errors=None*, *not_called_msg=None*, *nb_args_msg=None*, *other_args_msg=None*, *arg_names_msg=None*, *arg_defaults_msg=None*, *wrong_result_msg=None*, *wrong_output_msg=None*, *no_error_msg=None*, *wrong_error_msg=None*, *expand_message=True*, *state=None*)

Test a function definition.

**This function helps you test a function definition. Generally four things can be tested:**

1. The argument names of the function (including if the correct defaults are used)
2. The body of the functions (does it output correctly, are the correct functions used)
3. The return value with a certain input
4. The output value with a certain input
5. Whether certain inputs generate an error and what type of error

Custom feedback messages can be set for all these parts, default messages are generated automatically if none are set.

**Parameters**

- **`name`** (*`str`*) – the name of the function definition to be tested.
- **`arg_names`** (*`bool`*) – if True, the argument names will be tested, if False they won't be tested. Defaults to True.
- **`arg_defaults`** (*`bool`*) – if True, the default values of the arguments will be tested, if False they won't be tested. Defaults to True.
- **`body`** – this arguments holds the part of the code that will be ran to check the body of the function definition. It should be passed as a lambda expression or a function. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the body of the for loop. Defaults to None.
- **`results`** (*`list(list)`*) – a list of lists representing arguments that should be passed to the defined function. These arguments are passed to the function in the student environment and the solution environment, the results (what's returned) are compared.
- **`outputs`** (*`list(list)`*) – a list of lists representing arguments that should be passed to the defined function. These arguments are passed to the function in the student environment and the solution environment, the outpus are compared.
- **`errors`** (*`list(list)`*) – a list of lists representing arguments that should be passed to the defined function. These arguments are passed to the function in the student environment and the solution environment, the errors they generate are compared.
- **`not_called_msg`** (*`str`*) – message if the function is not defined.
- **`nb_args_msg`** (*`str`*) – message if the number of arguments do not matched.
- **`arg_names_msg`** (*`str`*) – message if the argument names do not match.
- **`arg_defaults_msg`** (*`str`*) – message if the argument default values do not match.

- **wrong_result_msg** (*str*) – message if one of the tested function calls' result did not match.
- **wrong_output_msg** (*str*) – message if one of the tested functions calls' output did not match.
- **no_error_msg** (*str*) – message if one of the tested function calls' result did not generate an error.
- **wrong_error_msg** (*str*) – message if the error that one of the tested function calls generated did not match.
- **expand_message** (*bool*) – only relevant if there is a body test. If True, feedback messages defined in the body test will be preceded by 'In your definition of ___, '. If False, *test_function_definition()* will generate no extra feedback if the body test fails. Defaults to True.

**Example** Student code:

```
def shout( word, times = 3):
    shout_word = not_word + '???'
    print( shout_word )
    return word * times
```

Solution code:

```
def shout( word = 'help', times = 3 ):
    shout_word = word + '!!!'
    print( shout_word )
    return word * times
```

SCT:

```
test_function_definition('shout')                          # fail
test_function_definition('shout', arg_defaults = False)    # pass
test_function_definition('shout', arg_defaults = False,    # fail
                                outputs = [('help')])

test_function_definition('shout', arg_defaults = False,    # pass
                                results = [('help', 2)])

test_function_definition('shout', args_defaults = False    # pass
        body = test_function('print', args = []]))
```

```
def test_function_definition(name,
                             arg_names=True,
                             arg_defaults=True,
                             body=None,
                             results=None,
                             outputs=None,
                             errors=None,
                             not_called_msg=None,
                             nb_args_msg=None,
                             other_args_msg=None,
                             arg_names_msg=None,
                             arg_defaults_msg=None,
                             wrong_result_msg=None,
                             wrong_output_msg=None,
                             no_error_msg=None,
                             wrong_error_msg=None,
                             expand_message=True):
```

In more advanced courses, you'll sometimes want students to define their own functions. With `test_function_definition()` it is possible to test such user-defined functions in a robust way. This function allows you to test four things:

1. The argument names of the function (including if the correct defaults are used)
2. The body of the functions (does it output correctly, are the correct functions used)
3. The return value with a certain input
4. The output value with a certain input

### Example 1

Say you want a student to write a very basic function to set numbers in a base from 1 up until 9 to a decimal. To not overcomplicate things you just ask them to implement the basic functionality; they don't have to catch any exceptions. A solution to the exercise can like like this:

~~~
*** =solution
```{python}
def to_decimal(number, base = 2):
    print("Converting %d from base %s to base 10" % (number, base))
    number_str = str(number)
    number_range = range(len(number_str))
    multipliers = [base ** ((len(number_str) - 1) - i) for i in number_range]
    decimal = sum([int(number_str[i]) * multipliers[i] for i in number_range])
    return decimal
```
~~~

You could test the function like this:

~~~
*** =sct
```{python}
# All of the following test_function_definition() functions are done on the same
# function definition.

# Test the function, see that the defaults of the arguments are the same.
# For this function, we don't care about the argument names of the function.
# Note: generally, we DO care about the names of the arguments, since they can
# be used as keywords. arg_defaults and arg_names will be set to True by default.

test_function_definition("to_decimal", arg_defaults = True, arg_names = False)

# Here, a feedback message will be generated. You can overwrite this feedback
# message by using:
# test_function_definition("to_decimal", arg_defaults = True, arg_names = False,
#     arg_defaults_msg = "Use the correct default argument values!")
# In the following tests, I'll always use the standard feedback messages, remember␣
↪they
# can almost always be overwritten.

# We want to test whether the function returns the correct things with certain inputs.

test_function_definition("to_decimal", arg_names = False, arg_defaults = False, #␣
↪Already tested this
~~~

```
    results = [
        [1001101, 2],
        ]1212357, 8]
)

# This will run to_decimal(1001101, 2) and to_decimal(1212357, 8) in student and␣
↪solution
# process, and match the results. If they don't match, a feedback message will be␣
↪generated.
# Note: here we've set arg_defaults to False, because we already tested this in the␣
↪first
# test_function_definition.

# We want to test the output of the function with certain inputs.

test_function_definition("to_decimal", arg_names = False, arg_defaults = False, #␣
↪Already tested this
    outputs = [
        [1234, 6],
        [8888888, 9]
)

# This will run to_decimal(1234, 6) and to_decimal(8888888, 9) in solution and student
# process and compare their printed output.

# Finally, we might want them to use a certain function. For this we can do tests␣
↪specifically
# on the body of the function. Remember you can use lambda functions or custom␣
↪functions for this
# (also see wiki about test_if_else(), test_for_loop() and test_while_loop().

test_function_definition("to_decimal", arg_names = False, arg_defaults = False, #␣
↪Already tested this
    body = lambda: test_function("sum", args = [], incorrect_msg = "you should use␣
↪the `sum()` function."))

# This will test the body of the function definition, and see if the function sum()␣
↪is used.
# Note that the generated feedback will be preceded by: 'In your definition of `to_
↪decimal()`, ...'
# So if the last test doesn't pass, this feedback will be generated:
#     In your definition of `to_decimal()`, you should use the `sum()` function.
```

```

Pitfall: you have to watch out when using `test_function()` in a body test, you should never test arguments that are only defined within the scope of the function (e.g. function parameters). This is the reason why we used `args = []` in the last test, because the argument used in `sum()` can not be calculated to verify in the global scope. This is something which would require architectural changes in the `pythonwhat` package.

### Example 2: User-defined errors

In some cases, you'll want the student to code resilience against incorrect inputs or behavior. To test this, you can use the `errors`, `no_error_msg` and `wrong_error_msg` arguments. The first is similar to `results`, and specifies the input arguments as a list of tuples or a list of lists, that have to generate an error. With `no_error_msg` you can control the message that is presented if running one of these argument sets does not generate an error, while it should. With `wrong_error_msg`, you control the message that is presented if the type of the error (or exception) that is
```

thrown does not correspond to the type that is thrown when the function is called in the solution process.

Suppose you want the student to code up a function `inc`, that increments a number if it's positive. If it's not, you want the function to raise a `ValueError`. A solution could look like this:

~~~
*** =solution
```{python}
def inc(num):
    if num < 0:
        raise ValueError('num is negative')
    return(num + 1)
```
~~~

To test this, we can use the following SCT (we're only focussing on the `errors` part here; of course you can extend the `test_function_definition()` call with more checks on arguments, `results`, body, etc.):

~~~
*** =sct
```{python}
test_function_definition("inc", errors = [[-1]])
```
~~~

If the student submits the following code:

```python
def inc(num):
    return(num + 1)
```

the SCT will see it's incorrect and throw the message: *Calling `inc(-1)` doesn't result in an error, but it should!*

If the student submits the following code:

```python
def inc(num):
    if num < 0:
        raise NameError('num is negative')
    return(num + 1)
```

the SCT will see it's incorrect and throw the message: *Calling `inc(-1)` should result in a `ValueError`, instead got a `NameError`.*

Currently, there isn't a way to test the actual message you pass with errors you raise.

### Example 3: `*args` and `**kwargs`

When defining a function in Python, it also possible to specify so-called 'unordered non-keyword arguments', with a `*`, and 'unordered keyword arguments'. Typically, these are called `args` and `kwargs` respectively, but this is not required.

Have a look at the following example:

~~~
*** =solution
```{python}
def my_fun(x, y = 4, z = ['a', 'b'], *args, **kwargs):
    k = len(args)
    l = len(kwargs)
    print("just checking")
    return k + l
```
~~~

An SCT to check this function definition:

~~~
*** =sct
```{python}
def inner_test():
    context = ['r', 's', ['c', 'd'], ['t', 'u'], {'a': 2, 'b': 3, 'd':4}]
    test_object_after_expression('k', context_vals = context)
    test_object_after_expression('l', context_vals = context)
test_function_definition("my_fun", body = inner_test,
        results = [{'args': ['r', 's', ['c', 'd'], 't', 'u', 'v'], 'kwargs': {'a': 2,
↪'b': 3, 'd': 4}}],
        outputs = [{'args': ['r', 's', ['c', 'd'], 't', 'u', 'v'], 'kwargs': {'a': 2,
↪'b': 3, 'd': 4}}])
```
~~~

There are different things to note:

- By default, the names of the `*` argument and the `**` argument are checked, if they are defined in the solution. This is controlled through `arg_names`, just like for 'regular' arguments. To override the automatic message that is thrown if the `*` or `**` arg is not specified or not appropriately named, use `other_args_msg`.
- The `*` and `**` args are also part of the context values that you can specify in 'inner tests'. They are appended to the normal arguments: first the `*`, then the `**` argument. You can see in the `context` object, that the penultimate element is used to specify the `*args` argument, and the last element, a dictionary, is used to specify the `**` argument.
- Before, you saw that `results`, `outputs`, and `errors` should be a list of lists, where the inner list is the list of arguments. To also cater for explicitly keyworded arguments, you can also specify a list of dictionaries. Each dictionary represents one call of the user-defined fucntion and should contain two elements: `'args'` and `'kwargs'`. Behind the scenes, the function will be called as: `my_fun([*d['args'], **d['kwargs']])`, where `d` is the two-key dictionary.

### Sidenote

Behind the scenes, `pythonwhat` has to fetch the value of objects from sub-processes. The required 'dilling' and 'undilling' can cause issues for exotic objects. For more information on this and possible errors that can occur, read the *Processes article*.

## test_function

**test_function**(*name*, *index=1*, *args=None*, *keywords=None*, *eq_condition='equal'*, *do_eval=True*, *not_called_msg=None*, *args_not_specified_msg=None*, *incorrect_msg=None*, *add_more=False*, *highlight=True*, *state=None*)

Test if function calls match.

This function compares a function call in the student's code with the corresponding one in the solution code. It will cause the reporter to fail if the corresponding calls do not match. The fail message that is returned will depend on the sort of fail.

**Parameters**

- **name** (*str*) – the name of the function to be tested.
- **index** (*int*) – index of the function call to be checked. Defaults to 1.
- **args** (*list(int)*) – the indices of the positional arguments that have to be checked. If it is set to None, all positional arguments which are in the solution will be checked.
- **keywords** (*list(str)*) – the indices of the keyword arguments that have to be checked. If it is set to None, all keyword arguments which are in the solution will be checked.

- **eq_condition** (*str*) – how arguments/keywords are compared. Currently, only "equal" is supported, meaning that the result in student and solution process should have exactly the same value.
- **do_eval** (*bool*) – True: arguments are evaluated and compared. False: arguments are not evaluated but 'string-matched'. None: arguments are not evaluated; it is only checked if they are specified.
- **not_called_msg** (*str*) – feedback message if the function is not called.
- **args_not_specified_msg** (*str*) – feedback message if the function is called but not all required arguments are specified
- **incorrect_msg** (*str*) – feedback message if the arguments of the function in the solution doesn't match the one of the student.

**Example** Student code:

```
import numpy as np
np.mean([1,2,3])
np.std([2,3,4])
```

Solution code:

```
import numpy
numpy.mean([1,2,3], axis = 0)
numpy.std([4,5,6])
```

SCT:

```
test_function("numpy.mean", index = 1, keywords = []) # pass
test_function("numpy.mean", index = 1)                # fail
test_function(index = 1, incorrect_op_msg = "Use the correct operators
↪")        # fail
test_function(index = 1, used = [], incorrect_result_msg = "Incorrect␣
↪result") # fail
```

```
test_function(name,
              index=1,
              args=None,
              keywords=None,
              eq_condition="equal",
              do_eval=True,
              not_called_msg=None,
              incorrect_msg=None)
```

`test_function()` enables you to test whether the student called a function correctly. The function first tests if the specified function is actually called by the student, and then compares the call with calls of the function in the solution code. Next, it can compare the parameters passed to these functions. Because `test_function()` also uses the student and solution process, this can be done in a very concise way.

## Example 1

Suppose you want the student to call the `round()` function on pi, as follows:

~~~
*** =solution
```{python}
# This is pi
~~~

```
pi = 3.14159

# Round pi to 3 digits
r_pi = round(pi, 3)
```
```

The following SCT tests whether the `round()` function is used correctly:

```
*** =sct
```{python}
test_function("round")
success_msg("Great job!")
```
```

This is a very robust way to test whether `round()` is used, much more robust when comparing to `test_student_typed()`. `test_function()` tests whether the student has called the function `round()` and checks whether the values of the arguments are the same as in the solution. So in this case, it tests wether `round()` is used with its first argument equal to `3.14159` and the second argument equal to `3`. `test_function()` figures out the values of these arguments from the solution code and the solution processes that corresponds with it. The above SCT would accept all of the following student submissions:

- `round(3.14159, 3)`
- `pi = 3.14159; dig = 3; round(pi, dig)`
- `int_part = 3; dec_part = 0.14159; round(int_part + dec_part, 3)`

By default, `test_function()` tests all arguments that are specified in the solution code. It is also possible to check whether a function is used and only check specific positional arguments. For example,

```
*** =sct
```{python}
test_function("round", args=[0])
success_msg("Great job!")
```
```

will only test whether the first argument's value is `3.14159`. A student submission that is `round(pi, 5)` would also pass this SCT.

With `args`, you can also control whether or not to actually check the values that were passed as parameters. Say you only want to check that the function `round()` was called:

```
*** =sct
```{python}
test_function("round", args=[])
success_msg("Great job!")
```
```

`test_function()` will automatically generate meaningful feedback, but you can also override these messages with `not_called_msg` and `incorrect_msg`. The former controls the message that is thrown if the student didn't call the specified function in the first place. The latter is thrown if the student did not correctly set the arguments in the function call:

```
*** =sct
```{python}
test_function("round",
              not_called_msg = "You did not call `round()` to round the irrational␣
↪number, `pi`.",
              incorrect_msg = "Be sure to round `pi` to `3` digits.`)
```

```
success_msg("Great job!")
```
```

### Example 2: Multiple function calls

`index`, which is 1 by default, becomes important when there are several calls of the same function. Suppose that your exercise requires the student to call the `round()` function twice: once on `pi` and once on `e`, Euler's number. A possible solution could be the following:

````
*** =solution
```{python}
# Call round on pi
round(3.14159, 3)

# Call round on e
round(2.71828, 3)
```
````

To test both these function calls, you'll need the following SCT:

````
*** =sct
```{python}
test_function("round", index=1)
test_function("round", index=2)
success_msg("Two in a row, great!")
```
````

The first `test_function()` call, where `index=1`, checks the solution code for the first function call of `round()`, finds it - `round(3.14159, 3)` - and then goes to look through the student code to find a function call of `round()` that matches the arguments. It is perfectly possible that there are 5 function calls of `round()` in the student's submission, and that only the fourth call matches the requirements for `test_function()`. As soon as a function call is found in the student code that passes all tests, `pythonwhat` heads over to the second `test_function()` call, where `index=2`. The same thing happens: the second call of `round()` is found from the solution code, and a match is sought for in the student code. This time, however, the function call that was matched before is now 'blacklisted'; it is not possible that the same function call in the student code causes both `test_function()` calls to pass.

This means that all of the following student submissions would be accepted:

- `round(3.14159, 3); round(2.71828, 3)`
- `round(2.71828, 3); round(3.14159, 3)`
- `round(3.14159, 3); round(123.456); round(2.71828, 3)`
- `round(2.71828, 3); round(123.456); round(3.14159, 3)`

Of course, you can also specify all other arguments to customize your test, such as `do_eval`, `args`, `not_called_msg` and `incorrect_msg`.

### Example 3: Custom feedback

By default `test_function()` checks all arguments and keywords that are specified in the solution; if you specify `incorrect_msg`, any error to one of these arguments will replaced by the same custom message. If you want to provide different custom error messages for different arguments, you can do so with multiple function calls. To, for example, provide different feedback for the first and second argument of the `round()` function:
````

~~~
*** =sct
```{python}
test_function("round", args = [0], index=1, incorrect_msg = 'first arg wrong!')
test_function("round", args = [1], index=1, incorrect_msg = 'second arg wrong!')
success_msg("Well done")
```
~~~

**NOTE**: currently, `test_function()` automatically checks all arguments and keywords that you specify in corresponding function call in the solution. Therefore, if you want to give specific feedback, make sure to select a single argument or a single keyword. To check the first argument, you can best use `args = [0], keywords = []`, to test a keyword named `check`, you'll want to use `args = [], keywords = ['check']`.

### Example 4: Methods

Python also features methods, i.e. functions that are called on objects. For testing such a thing, you can also use `test_function()`. Consider the following solution code, that creates a connection to an SQLite Database with `sqlalchemy`.

~~~
*** =solution
```{python}
from urllib.request import urlretrieve
from sqlalchemy import create_engine, MetaData, Table
engine = create_engine('sqlite:///census.sqlite')
metadata = MetaData()
connection = engine.connect()
from sqlalchemy import select
census = Table('census', metadata, autoload=True, autoload_with=engine)
stmt = select([census])

# execute the query and fetch the results.
connection.execute(stmt).fetchall()
```
~~~

To test the last chained method calls, you can use the following SCT. Notice from the second `test_function()` call here that you have to describe the entire chain (leaving out the arguments that are passed to `execute()`). This way, you explicitly list the order in which the methods should be called.

~~~
*** =sct
```
test_function("connection.execute", do_eval = False)
test_function("connection.execute.fetchall")
```
~~~

**NOTE**: currently, it is not possible to easily test the arguments inside chained method calls, methods inside arguments, etc. We are working on a massive update of `pythonwhat` to easily support this very customized testing, with virtually no limit to 'how deep you want the tests to go'. More on this later!

#### `do_eval`

With `do_eval`, you can control how arguments are compared between student and solution code.

- If `do_eval` is `True`, the evaluated version of the arguments are compared;
- If `do_eval` is `False`, the 'string version' of the argumetns are compared;

- If `do_eval` is `None`, the arguments are not compared; in this case, `test_function()` simply checks if you specified the arguments, without further checks.

### Function calls in packages

If you're testing whether function calls of particular packages are used correctly, you should always refer to these functions with their 'full name'. Suppose you want to test whether the function `show` of `matplotlib.pyplot` was used correctly, you should use

~~~
*** =sct
```{python}
test_function("matplotlib.pyplot.show")
```
~~~

The `test_function()` call can handle it when a student used aliases for the python packages (all `import` and `import * from *` calls are supported). In case there is an error, `test_function()` will automatically generated a feedback message that uses the alias of the student.

**NOTE:** No matter how you import the function, you always have to refer to the function with its full name, e.g. `package.subpackage1.subpackage2.function`.

### Argument equality

Just like with `test_object()`, evaluated arguments are compared using the == operator (check out the section about Object equality). For a lot of complex objects, the implementation of == causes the object instances to be compared... not their underlying meaning. For example when the solution is:

```
*** =solution
from urllib.request import urlretrieve
fn1 = 'https://s3.amazonaws.com/assets.datacamp.com/production/course_998/datasets/
↪Chinook.sqlite'
urlretrieve(fn1, 'Chinook.sqlite')

# Import packages
from sqlalchemy import create_engine
import pandas as pd

# Create engine: engine
engine = create_engine('sqlite:///Chinook.sqlite')

# Execute query and store records in dataframe: df
df = pd.read_sql_query("SELECT * FROM Album", engine)
```

And the SCT is:

```
*** =sct
test_function("pandas.read_sql_query")
```

The SCT will fail even if the student uses this exact solution code. The reason being that the `engine` object is compared in the solution and student process. The engine object is evaluated by `create_engine('sqlite:///Chinook.sqlite')`. As you can try out yourself, `create_engine('sqlite:///Chinook.sqlite') == create_engine('sqlite:///Chinook.sqlite')` will always be `False`, even though they are semantically exactly the same. A better way of testing this code would be:

```
*** =sct
test_correct(
    lambda: test_object("df"),
    lambda: test_function("pandas.read_sql_query", do_eval=False)
)
```

This SCT will not do exactly the same, but it will test enough in practice 99% of the time. Check out the section about Object equality for complex objects that do have a good equality implementation.

**NOTE**: Behind the scenes, `pythonwhat` has to fetch the value of objects from sub-processes. The required 'dilling' and 'undilling' can cause issues for exotic objects. For more information on this and possible errors that can occur, read the Processes article.

## test_function_v2

**`test_function_v2`**(*name*, *index=1*, *params=[]*, *signature=None*, *eq_condition='equal'*, *do_eval=True*, *not_called_msg=None*, *params_not_matched_msg=None*, *params_not_specified_msg=None*, *incorrect_msg=None*, *add_more=False*, *highlight=True*, *state=None*)

Test if function calls match (v2).

This function compares a function call in the student's code with the corresponding one in the solution code. It will cause the reporter to fail if the corresponding calls do not match. The fail message that is returned will depend on the sort of fail.

**Parameters**

- **name** (*str*) – the name of the function to be tested.
- **index** (*int*) – index of the function call to be checked. Defaults to 1.
- **params** (*list (str)*) – the parameter names of the function call that you want to check.
- **signature** (*Signature*) – Normally, test_function() can figure out what the function signature is, but it might be necessary to use build_sig to manually build a signature and pass this along.
- **eq_condition** (*str*) – how parameters are compared. Currently, only "equal" is supported, meaning that the arguments in student and solution process should have exactly the same value.
- **do_eval** (*list (bool)*) – Boolean or list of booleans (parameter-specific) that specify whether or not arguments should be evaluated. True: arguments are evaluated and compared. False: arguments are not evaluated but 'string-matched'. None: arguments are not evaluated; it is only checked if they are specified.
- **not_called_msg** (*str*) – custom feedback message if the function is not called.
- **params_not_matched_message** (*str*) – custom feedback message if the function parameters were not successfully matched.
- **params_not_specified_msg** (*str*) – string or list of strings (parameter-specific). Custom feedback message if not all parameters listed in params are specified by the student.
- **incorrect_msg** (*list (str)*) – string or list of strings (parameter-specific). Custom feedback messages if the arguments don't correspond between student and solution code.

```
test_function_v2(name,
                 index=1,
                 params=None,
                 signature=None,
                 eq_condition="equal",
                 do_eval=True,
                 not_called_msg=None,
                 params_not_matched_msg=None,
                 params_not_specified_msg=None,
                 incorrect_msg=None)
```

`test_function_v2()` enables you to test whether the student called a function correctly. The function first tests if the specified function is actually called by the student, and then compares the call with calls of the function in the solution code. Next, it can compare the arguments passed to these functions. Because `test_function_v2()` also uses the student and solution process, this can be done in a very concise way. `test_function_v2()` is an improved version of `test_function()`, where:

- there is resilience against different ways of calling a function (arguments vs keywords),
- you have to be specific about which parameters you want to check,
- you can specify parameter-specific evaluation forms (`do_eval` can be a list),
- you can specify parameter-specific custom messages (`params_not_matched_msg` and `params_not_specified_msg` can be lists),
- you have more control over messaging in general.

### Example 1

Suppose you want the student to call the `round()` function on pi, as follows:

~~~
*** =solution
```{python}
# This is pi
pi = 3.14159

# Round pi to 3 digits
r_pi = round(pi, 3)
~~~
```

The following SCT tests whether the `round()` function is used correctly:

```
*** =sct
```{python}
test_function_v2("round", params=["number", "ndigits"])
success_msg("Great job!")
```
```

`test_function_v2()` tests whether the student has called the function `round()` and checks whether the values of the arguments are the same as in the solution. So in this case, it tests whether `round()` is used and the `number` and `ndigits` parameters, that `round()` expects, are specified correctly, i.e. equal to `3.14159` and `3` respectively. `test_function_v2()` figures out the values of these arguments from the solution code and the solution process that corresponds with it. The above SCT would accept all of the following student submissions:

- `round(3.14159, 3)`
- `round(number=3.14159, 3)`
- `round(number=3.14159, ndigits=3)`
- `round(ndigits=3, number=3.14159)`
```

- `pi=3.14159; dig=3; round(pi, dig)`
- `pi=3.14159; dig=3; round(number=pi, dig)`
- `int_part = 3; dec_part = 0.14159; round(int_part + dec_part, 3)`

In `params`, you have to explicitly list all the parameters that you want to test. If you only want to check the `number` parameter, for example, you can use:

~~~
*** =sct
```{python}
test_function_v2("round", params=["number"])
success_msg("Great job!")
~~~
```

This SCT will only test whether the `number` parameter was specified to be `3.14159`. If a student submits `round(pi, 5)`, this would also pass this SCT.

If you specify `params` to be an empty list, which is the default, you are simply checking whether the `round()` function was called in the first place:

```
*** =sct
```{python}
test_function_v2("round") # same as test_function_v2("round", params=[])
success_msg("Great job!")
```
```

`test_function_v2()` will automatically generate meaningful feedback, but you can also override these messages through the different `*_msg` parameters that `test_function_v2()` features:

- `not_called_msg`: message if the student didn't call the specified function or didn't call the specified function often enough (if you're testing multiple calls of the same function in the same submission).
- `params_not_matched_msg`: message if the function call of the student was invalid, i.e. if the way of specifying the different parameters was invalid.
- `params_not_specified_msg`: message if the student did not specify all parameters that are specified inside `params`. This argument can either be a string, to give the same message for each parameter that is missing, or a list of strings with the same length as `params`. In case of a missing parameter, `test_function_v2()` will present the corresponding message.
- `incorrect_msg`: message if the student did not specify all parameters correctly, so when his or her specifications don't correspond with the solution. This argument can again be a single string, or a list of parameter-specific feedback messages.

Below is an example of an SCT that specified all feedback messages. This is not required, though; you can depend on the automatic feedback messages for the `not_called_msg`, `params_not_specified_msg` and `incorrect_msg` and only manually specify the `params_not_matched_msg`, for example.

```
*** =sct
```{python}
test_function_v2("round", params=["number", "ndigits"]
                 not_called_msg="You did not call `round()` to round the irrational␣
↪number, `pi`.",
                 params_not_matched_msg="Are you sure you correctly called the␣
↪`round()` function?",
                 params_not_specified_msg="Make sure to specify both the `number` and␣
↪`ndigits` parameter!",
                 incorrect_msg=["Make sure to correctly specify `number`; it should␣
↪be `pi`, or `3.14159`.",
                                "Have you specified `ndigits` so that `pi` is rounded␣
↪to 3 digits?"])
```

```
success_msg("Great job!")
```
```

### Example 2: Multiple function calls

`index`, which is 1 by default, becomes important when there are several calls of the same function. Suppose that your exercise requires the student to call the `round()` function twice: once on `pi` and once on `e`, Euler's number. A possible solution could be the following:

````
*** =solution
```{python}
# Call round on pi
round(3.14159, 3)

# Call round on e
round(2.71828, 3)
```
````

To test both these function calls, you'll need the following SCT:

````
*** =sct
```{python}
test_function_v2("round", params=["number","ndigits"], index=1)
test_function_v2("round", params=["number","ndigits"], index=2)
success_msg("Two in a row, great!")
```
````

The first `test_function_v2()` call, where `index=1`, checks the solution code for the first function call of `round()`, finds it - `round(3.14159, 3)` - and then goes to look through the student code to find a function call of `round()` that matches the arguments. It is perfectly possible that there are 5 function calls of `round()` in the student's submission, and that only the fourth call matches the requirements for `test_function_v2()`. As soon as a function call is found in the student code that passes all tests, `pythonwhat` heads over to the second `test_function_v2()` call, where `index=2`. The same thing happens: the second call of `round()` is found from the solution code, and a match is sought for in the student code. This time, however, the function call that was matched before is now 'blacklisted'; it is not possible that the same function call in the student code causes both `test_function_v2()` calls to pass.

This means that all of the following student submissions would be accepted:

- `round(3.14159, 3); round(2.71828, 3)`
- `round(2.71828, 3); round(3.14159, 3)`
- `round(number=3.14159, ndigts=3); round(number=2.71828, 3)`
- `round(number=2.71828, 3); round(number=3.14159, 3)`
- `round(3.14159, 3); round(123.456); round(2.71828, 3)`
- `round(2.71828, 3); round(123.456); round(3.14159, 3)`

Of course, you can also specify all other arguments to customize your test to perfection, such as custom messages and `do_eval` (example 3).
````

### Example 3: `do_eval`

With `do_eval`, you can control how parameter specifications are compared between student and solution code. There are two ways to specify `do_eval`: you can specify a single value, that will be used for comparing all `params` that you specified. However, you can also specify a list of values, with the same length as `params`; the way in which parameter specifications are compared becomes parameter specific. In both cases, there are three valid values:

- `True`, where the evaluated version of the student and solution arguments is compared.
- `False`, where the 'string version' of the arguments is compared;
- `None`, in which case the arguments are not compared; `test_function_v2` simply checks if the parameter(s) in question has/have been specified.

Say, for example, you want to check if a student called the `round()` function and specified the parameters `number` and `ndigits`. You want to test the actual equality of `number`, but you don't care about the value of `ndigits`, you just want to make sure the student specified it, nothing more.

The following solution and SCT implement this train of thought (custom feedback messages have not been specified, although this is perfectly possible):

~~~
*** =solution
```{python}
# This is pi
pi = 3.14159

# Round pi to 3 digits
r_pi = round(pi, 3)
```

*** =sct
```{python}
test_function_v2("round",
                 params=["number", "ndigits"],
                 do_eval=[True, None])
success_msg("Great job!")
```
~~~

All of the following submissions would be accepted by this SCT:

- `round(pi, 3)`
- `round(number=pi, ndigits=3)`
- `round(number=pi, ndigits=4)`
- `round(pi, 4)`
- `round(pi, 0)`

### Example 4: Function calls in packages

If you're testing whether function calls of particular packages are used correctly, you should always refer to these functions with their 'full name'. Suppose you want to test whether the function `show` of `matplotlib.pyplot` was used correctly, you should use

~~~
*** =sct
```{python}
test_function_v2("matplotlib.pyplot.show")
```
~~~

The `test_function_v2()` call can handle it when a student used aliases for the python packages (all `import` and `import * from *` calls are supported). In case there is an error, `test_function_v2()` will automatically generated a feedback message that uses the alias that the student used.

**NOTE:** No matter how you import the function, you always have to refer to the function with its full name, e.g. `package.subpackage1.subpackage2.function`.

### Example 5: Manual signatures

To implement resilience against different ways of specify function parameters, the `inspect` module is used, that is part of Python's basic distribution. Through `inspect.signature()` a function's parameters can be inferred, and then 'bound' to the arguments that the student specified. However, this signature is not available for all of Python's functions. More specifically, Python's built-in functions that are implemented in C don't allow a signature to be extracted from them. `pythonwhat` already includes manually specified signatures for functions such as `print()`, `str()`, `hasattr()`, etc, but it's still possible that some signatures are missing.

That's why `test_function_v2()` features a `signature` parameter, that is `None` by default. If `pythonwhat` can't retrieve a signature for the function you want to test, you can pass an object of the class `inspect.Signature` to the `signature` parameter.

Suppose, for the sake of example, that `test_function_v2()` can't find a signature for the `round()` function (you will be informed by this through automated testing; running the solution against an SCT that depends on a signature that is not found will throw a backend error). To be able to implement this function test, you can use the `sig_from_params()` function:

~~~
*** =sct
```{python}
sig = sig_from_params(param("number", param.POSITIONAL_OR_KEYWORD),
                      param("ndigits", param.POSITIONAL_OR_KEYWORD, default=0))
test_function_v2("round", params=["number", "ndigits"], signature=sig)
```
~~~

`param` is an alias of the `Parameter` class that's inside the `inspect` module. You can pass `sig_from_params()` as many parameters as you want. The first argument of `param()` should be the name of the parameter, the second argument should be the 'kind' of parameter. `param.POSITIONAL_OR_KEYWORD` tells `test_function_v2` that the parameter can be specified either through a positional argument or through a keyword argument. Other common possibilities are `param.POSITIONAL_ONLY` and `param.KEYWORD_ONLY` (for a full list, refer to the Python docs on `inspect`). The third, optional argument, allows you to specify a default value for the parameter.

**NOTE:** If you find vital Python functions that are used very often and that are not included in `pythonwhat` by default, you can let us know and we'll add the function to our list of manual signatures.

### Example 6: Methods

Python also features methods, i.e. functions that are called on objects. For testing such a thing, you can also use `test_function_v2()`. Consider the following solution code, that creates a connection to an SQLite Database with `sqlalchemy`.

~~~
*** =solution
```{python}
# Prepare everything
~~~

from urllib.request import urlretrieve from sqlalchemy import create_engine, MetaData, Table engine = create_engine('sqlite:///census.sqlite') metadata = MetaData() connection = engine.connect() from sqlalchemy import select census = Table('census', metadata, autoload=True, autoload_with=engine) stmt = select([census])

```
# execute the query and fetch the results.
connection.execute(stmt).fetchall()
```
```

To test the last chained method calls, you can use the following SCT. Notice from the second `test_function_v2()` call here that you have to describe the entire chain (leaving out the arguments that are passed to `execute()`). This way, you explicitly list the order in which the methods should be called.

````
*** =sct
```{python}
test_function_v2("connection.execute", params = ["object"], do_eval = False)
test_function_v2("connection.execute.fetchall")
```
````

**NOTE**: currently, it is not possible to easily test the arguments inside chained method calls, methods inside arguments, etc. We are working on a massive update of `pythonwhat` to easily support this very customized testing, with virtually no limit to 'how deep you want the tests to go'. More on this later!

### Example 7: Signatures for methods

In the previous example, you might have noticed that `test_funtion_v2()` was capable to infer that `connection` is a `Connection` object, and that `execute()` is a method of the `Connection` class. For checking method calls that aren't chained, this is possible, but for chained method calls, such as `connection.execute.fetchall`, this is not possible. In those cases you'll have to manually specify a signature. With `sig_from_obj()` you can specify the function from which to extract a signature.

The following full example shows how it's done:

````
*** =pre_exercise_code
```{python}
class Test():
    def __init__(self, a):
        self.a = a

    def set_a(self, value):
        self.a = value
        return(self)
x = Test(123)
```

*** =solution
```{python}
x.set_a(843).set_a(102)
```

*** =sct
```{python}
sig = sig_from_obj('x.set_a')
test_function_v2('x.set_a.set_a', params=['value'], signature=sig)
```
````

**NOTE**: You can also use the `sig_from_params()` function to manually build the signature from scratch, but this this more work than simply specifying the function object as a string from which to extract the signature.
````

### Extra: Argument equality

Just like with `test_object()`, evaluated arguments are compared using the == operator (check out the section about Object equality). For a lot of complex objects, the implementation of == causes the object instances to be compared... not their underlying meaning. For example when the solution is:

```
*** =solution
```
from urllib.request import urlretrieve
fn1 = 'https://s3.amazonaws.com/assets.datacamp.com/production/course_998/datasets/
↪Chinook.sqlite'
urlretrieve(fn1, 'Chinook.sqlite')
from sqlalchemy import create_engine
import pandas as pd
engine = create_engine('sqlite:///Chinook.sqlite')

# Execute query and store records in dataframe: df
df = pd.read_sql_query("SELECT * FROM Album", engine)
```
```

And the SCT is:

```
*** =sct
```
test_function_v2("pandas.read_sql_query", params = ['sql', 'con'], do_eval = [True,␣
↪False])
```
```

The SCT will fail even if the student uses this exact solution code. The reason being that the `engine` object is compared in the solution and student process. The engine object is evaluated by `create_engine('sqlite:///Chinook.sqlite')`. As you can try out yourself, `create_engine('sqlite:///Chinook.sqlite') == create_engine('sqlite:///Chinook.sqlite')` will always be `False`, even though they are semantically exactly the same. A better way of testing this code would be:

```
*** =sct
test_correct(
    lambda: test_object("df"),
    lambda: test_function_v2("pandas.read_sql_query", do_eval=False)
)
```

This SCT will not do exactly the same, but it will test enough in practice 99% of the time. Check out the section about Object equality for complex objects that DO have a good equality implementation.

**NOTE**: Behind the scenes, `pythonwhat` has to fetch the value of objects from sub-processes. The required 'dilling' and 'undilling' can cause issues for exotic objects. For more information on this and possible errors that can occur, read the *Processes article*.

## test_lambda_function

**test_lambda_function**(*index*, *arg_names=True*, *arg_defaults=True*, *body=None*, *results=[]*, *errors=[]*, *not_called_msg=None*, *nb_args_msg=None*, *arg_names_msg=None*, *arg_defaults_msg=None*, *wrong_result_msg=None*, *no_error_msg=None*, *expand_message=True*, *state=None*)

Test a lambda function definition.

**This function helps you test a lambda function definition. Generally four things can be tested:**

1. The argument names of the function (including if the correct defaults are used)
2. The body of the functions (does it output correctly, are the correct functions used)
3. The return value with a certain input
4. Whether certain inputs generate an error

Custom feedback messages can be set for all these parts, default messages are generated automatically if none are set.

**Parameters**

- **index** (*int*) – the number of the lambda function you want to test.
- **arg_names** (*bool*) – if True, the argument names will be tested, if False they won't be tested. Defaults to True.
- **arg_defaults** (*bool*) – if True, the default values of the arguments will be tested, if False they won't be tested. Defaults to True.
- **body** – this arguments holds the part of the code that will be ran to check the body of the function definition. It should be passed as a lambda expression or a function. The functions that are ran should be other pythonwhat test functions, and they will be tested specifically on only the body of the for loop. Defaults to None.
- **results** (*list (str)*) – a list of strings representing function calls to the lam function. The lam function will be replaced by the actual lambda function from the student and solution code. The result of calling the lambda function will be compared between student and solution.
- **errors** (*list (str)*) – a list of strings representing function calls to the lam function. The lam function will be replaced by the actual lambda function from the student and solution code. It will be checked if an error is generated appropriately for the specified inputs.
- **not_called_msg** (*str*) – message if the function is not defined.
- **nb_args_msg** (*str*) – message if the number of arguments do not matched.
- **arg_names_msg** (*str*) – message if the argument names do not match.
- **arg_defaults_msg** (*str*) – message if the argument default values do not match.
- **wrong_result_msg** (*str*) – message if one of the tested function calls' result did not match.
- **no_error_msg** (*str*) – message if one of the tested function calls' result did not generate an error.
- **expand_message** (*bool*) – only relevant if there is a body test. If True, feedback messages defined in the body test will be preceded by 'In your definition of ___, '. If False, `test_function_definition()` will generate no extra feedback if the body test fails. Defaults to True.

```
def test_lambda_function(index,
                         arg_names=True,
                         arg_defaults=True,
                         body=None,
                         results=None,
                         errors=None,
                         not_called_msg=None,
                         nb_args_msg=None,
                         arg_names_msg=None,
                         arg_defaults_msg=None,
                         wrong_result_msg=None,
                         no_error_msg=None,
                         expand_message=True)
```

With `test_function_definition()`, you can only test user-defined functions that have a name. There is an important class of functions in Python that go by the name of lambda functions. These functions are anonymous, so they don't necessarily require a name. To be able to test user-coded lambda functions, the `test_lambda_function()` is available. If you're familiar with `test_function_definition()`, you'll notice some similarities. However, instead of the `name`, you now have to pass the `index`; this means you have to specify the lambda function definition to test by number (test the first, second, third ...). Also, because we don't necessarily have an object represents a lambda function (because it can be anonymous), some tricky things are required to correctly specify the arguments `errors` and `results`; the example will give more details.

### Example 1

Suppose we want the student to code a lambda function that takes two arguments, `word` and `echo`, the latter of which should have default value 1. The lambda function should return the product of `word` and `echo`. A solution to this challenge could be the following:

~~~
*** =solution
```{python}
echo_word = lambda word, echo = 1: word * echo
```
~~~

To test this lambda function definition, you can use the following SCT:

~~~
*** =sct
```
test_lambda_function(1,
                 body = lambda: test_student_typed('word'),
                 results = ["lam('test', 2)"],
                 errors = ["lam('a', '2')"])
```
~~~

With `1`, we tell `pythonwhat` to test the first lambda function it comes across. Through body, we can specify sub-SCTs to be tested on the body of the lambda function (similar to how `test_function_definition` does it). With `results` and `errors`, you can test the lambda function definition for different input arguments. Notice here that you have to specify a list of function calls as a string. The function you have to call is `lam()`; behind the scenes, this `lam` will be replaced by the actual lambda function the student and solution defined. This means that `lam('test', 2)` will be converted into:

~~~
```
(lambda word, echo = 1: word * echo)('test', 2)
```
~~~

That way, the system can run the function call, and compare the results between function and solution. Things work the same way for `errors`.

As usual, the `test_lambda_function()` will generate a bunch of meaningful automated messages depending on which error the student made (you can override all these messages through the `*_msg` argument):

```
submission: <empty>
feedback: "The system wants to check the first lambda function you defined but hasn't␣
↪found it."

submission: echo_word = lambda wrd: wrd * 1
```

```
feedback: "You should define the first lambda function with 2 arguments, instead got␣
↪1."

submission: echo_word = lambda wrd, echo: wrd * echo
feedback: "In your definition of the first lambda function, the first argument should␣
↪be called <code>word</code>, instead got <code>wrd</code>."

submission: echo_word = lambda word, echo = 2: word * echo
feedback: "In your definition of the first lambda function, the second argument␣
↪should have <code>1</code> as default, instead got <code>2</code>."

submission: echo_word = lambda word, echo = 1: 2 * echo
feedback: "In your definition of the first lambda function, could not find the␣
↪correct pattern in your code."

submission: echo_word = lambda word, echo = 1: word * echo + 1
feedback: "Calling the the first lambda function with arguments <code>('test', 2)</
↪code> should result in <code>testtest</code>, instead got an error."

submission: echo_word = lambda word, echo = 1: word * echo * 2
feedback = "Calling the first lambda function with arguments <code>('test', 2)</code>␣
↪should result in <code>testtest</code>, instead got <code>testtesttesttest"

submission: echo_word = lambda word, echo = 1: word * int(echo)
feedback: "Calling the first lambda function with the arguments <code>('a', '2')</
↪code> doesn't result in an error, but it should!"

submission: echo_word = lambda word, echo = 1: word * echo
feedback: "Great job!" (pass)
```

### What about testing usage?

This is practically impossible to do in a robust way; we suggest you do this in an indirect way (checking the output that should be generated, checking the object that should be created, etc).

**NOTE**: Behind the scenes, `pythonwhat` has to fetch the value of objects from sub-processes. The required ‘dilling’ and ‘undilling’ can cause issues for exotic objects. For more information on this and possible errors that can occur, read the *Processes article*.

# CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index