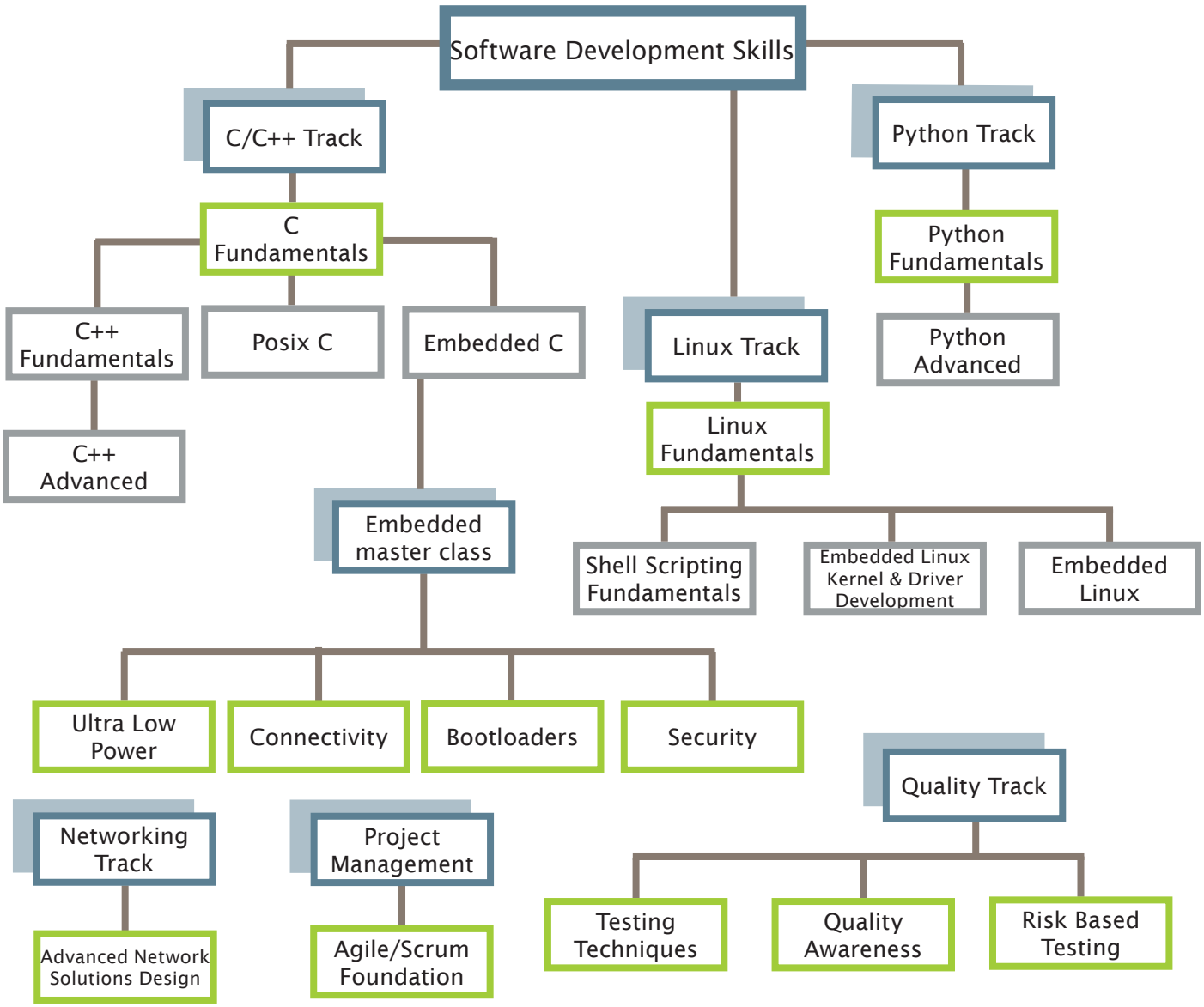
INTELLIGENT SYSTEMS ACADEMY



Check also:

picoTCP

The answer for a size, speed and feature conscious open source TCP/IP stack for embedded devices.

www.picotcp.com

github.com/tass-belgium/picotcp/

VueForge

A unique end-to-end offer that combines technology, usage and profitability with an in-depth industry understanding, key in-house accelerators and a supporting ecosystem.

http://intelligent-systems.altran.com/core-offers/vue-forge

Intelligent Systems | ALTRAN

1 Data Types

Objective: Get to know Python's data types

1.1 Calculations

Calcualte the following using the Python interactive interpreter :

- sum of 3 and 4+2j
- difference of 4+2j and 7-8j
- 1 devided by 3 without flooring
- 2 to the power of 8

1.2 Strings

Execute the following using the Python interactive interpreter:

- Initialize a string object with 'invent a cool string'
- Check the length of the string
- Get the 5th character in the string
- Get the 2nd to last character in the string
- Get a substring of the last 5 characters in the string
- Get the ASCII value of the first character in the string

1.3 Lists

Execute the following using the Python interactive interpreter:

- Initialize a string object with 'Monty Python'
- Create a list of individual characters of previous string using a list comprehension
- Append ' Rules' to the list character by character
- Create another list from a string ' big time', using a list comprehension
- Create a new list from the two previous lists

1.4 Dictionaries

Execute the following using the Python interactive interpreter:

- Create a dictionary that maps Belgian province capitals to their Postal code
- Get the code for Ghent
- Get the code for a non-capital city, like Ostend
- Get a list of the capitals in the dict
- Create a list of pairs (list of 2 items), that maps the postal code to the name, using the list of capitals and the dict in a list comprehension
- Create a dict from the list

1.5 Tuples

Execute the following using the Python interactive interpreter:

- Create a list of tuples where the first value in the tuple is a temperature in Celsius, and the second in Fahrenheit. Do this for -10 to 50
  - Take a look at the `range` function
- Samething, but places swapped. From 0 to 100 Fahrentheit.

Celsius = (Fahrenheit - 32) * 5/9

1.6 Strings

Files: string1.py, string2.py

We'll be using some basic Python exercises provided by Google. Download the "google-python-exercises.zip" from the developers.google.com website, or ask you instructor to share it with you.

Under the "basics" directory you'll find `string1.py` and `string2.py`. These files contain lots of code already, and you are invited to "fill the blanks" to pass the tests in the "main" function. The permissions should be set so you can execute like "`./string1.py`". Alternatively you can also "`python string1.py`".

1.7 Lists

Files: list1.py, list2.py

In the same "basics" directory of the Python exercises by Google, you'll find the `list{1|2}.py` files. Same approach as with Strings, complete the file, to make the tests pass.

2 Functions

2.1 Dictionary inverse

Files: dict_inverse.py
Objective: Write your first Python function

Write a python function that assumes a dict as argument. Create a dict where the values are the keys. In case of multiple values, make a list of the keys that had the value. Return the dictionary.

Your choice if you write a script or do this interactively. In any case, test your function in different scenarios.

Those of you who are serious about writing quality code might want to start with "Python unittests" right away. Without going in to detail, the following would be the skeleton code for such a unittest program all in one file. (We'll look at modules later, to separate into multiple files more nicely).

```
import unittest

def dict_inverse(d):
   <Your code>


class Test_dict_inverse(unittest.TestCase):
    def test_dict_inverse(self):
        d = { "Oostende" : 8400,
              "Zandvoorde" : 8400 ,
              "Stene" : 8400,
              "Brugge" : 8000,
              "Gent" : 9000
              }

        d2 = dict_inverse(d)

        self.assertEqual(len(d2), 3)

        self.assertEqual(len(d2[8400]), 3)
        for city in ("Oostende","Zandvoorde","Stene"):
            self.assertTrue(city in d2[8400])

        self.assertEqual(d2[8000], ["Brugge"])
        self.assertEqual(d2[9000], ["Gent"])
```

```
unittest.main()
```

2.2 Prime, Armstrong and Perfect number tester

Files: number_tester.py
Objective: Get comfortable with functions and maths in Python

Write a program containing three functions. Respectively to test an input for Prime, Armstrong or Perfect number.
For TASS'ers or brave customers: Use the skeleton code for unittesting from the previous exercise. The test class should now have 3 test_* functions. These tests should obviously test the functions with values that cover all aspects of the function.

To convert a string to number, use `number = int('3')` for example.

To generate a range of numbers, use `range(from, to)` in a for loop as such: `for d in range(1,10):`

2.2.1 Testing for Primes

Use the brute force approach. Use `range` to test each number in [2,n/2] as a divisor.

If you're comfortable enough with Python to try something off the beaten path, take any of the other available methods. (See Wikipedia)

2.2.2 Testing for Armstrong numbers

This one is a hybrid between math and string operations. An Armstrong or narcissistic number is the sum of all it's digits raised to the power of the number of digits in the number.

You'll need to convert all digits in the number from 1-digit string objects to int objects using the conversion given above.

$$153 = 1^3 + 5^3 + 3^3 \tag{1}$$

$$370 = 3^3 + 7^3 + 0^3 \tag{2}$$

2.2.3 Testing for Perfect numbers

A perfect number is a positive integer that is equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself.

Unlike with a Prime tester, where you might stop after the first divisor is found that isn't 1 or itself, you need to get all divisors and make the sum of them.

$$6 = 1 + 2 + 3 \tag{3}$$

$$28 = 1 + 2 + 4 + 7 + 14 \tag{4}$$

2.3 Palindrome tester

Files: palindrome.py
Objective: Learn more about comparing parts of strings

Write a program containing a function which returns True if a string is a palindrome. Write a unit test.

Test with palindromes like:

- ABBA
- SOS
- SOs
- Kayak
- Satanoscillatemymetallicsonatas

2.4 Matrix transposition

Files: transpose.py
Objective: Get comfortable with lists

Write a program containing a function which returns a transposed matrix. Have it accept non-square matrixes as well. Write a unit test.

3 Modules

3.1 Temperature Conversion

Files: temperature.py, convertor.py, test_temperature.py
Objective: Write your first module, write tests for it and measure coverage

3.1.1 Module

Write a module "temperature" which converts from Celsius to Fahrenheit and Kelvin and vice versa.

To get you underway:

- Kelvin = Celsius + 273.15
- Celsius = (Fahrenheit - 32) * 5/9;

Those who are brave enough can write their own unit tests. Others can just use the unittests from the solutions. They will either need to conform to that particular interface or refactor the unittests.

If you dont' want to implement all conversions + unittests, you're free to limit yourself to 2. But think about it, implementations are all one-liners most likely. And testing for 4 numbers you calculate using Google is also not much effort.

If you're skipping the unit tests, you might want to skip to the "Program" subsection. And write a program that accepts command line arguments like a real program, and uses the`temperature` module you wrote to do the actual calculations.

3.1.2 Unit Tests

Write UnitTests for this module in another script called `test_temperature.py`. Write your `temperature.py` using the same interface as the solutions, so you can run the solutions UnitTest against your module.

You might want to check into `assertAlmostEqual()` for this part.

Here's a table with some values you can use for unit testing:

Fahrenheit	Celsius
10	-12.2222
50	10
100	37.7778

Those who are serious about software engineering, might want to do this the TDD way!! (Write tests for one feature first, with some skeleton code, have the tests and skeleton run, but fail the tests, implement functionality until tests pass, start over with another feature...)

3.1.3 Coverage

Install the coverage tool with `pip install coverage`. If you're running a system with both Python2 and 3 you might want to do `easy_install3 pip` first and then `pip3 install coverage`. This assumes you're using Python3 by default ...

Run it as follows:

```
$ coverage run test_temperature.py
......
----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK
$ coverage report -m
Name               Stmts   Miss  Cover   Missing
--------------------------------------------------
temperature           41     26    37%   44-82
test_temperature      26      0   100%
--------------------------------------------------
TOTAL                 67     26    61%
```

You can also get a nice html output with `coverage html`. Visit the file `htmlcov/index.html` in your browser.

Needless to say: Get as much coverage as possible!!

3.1.4 Program

Write a `convertor.py` program that uses your `temperature` module for the actual calculations. Accept multiple conversions like below, by using `argparse`

```
./temperature.py -c2f 100 -f2c 90 -c2k -100
```

3.1.5 Coverage

Run the coverage tool again. Think about the results, and how you're going to remedy the situation. After giving it some thought, talk to your teacher about this ... Or take (no more) 5 minutes to go through the manual of Coverage.

3.2 Python UnitTest Skeleton generator

Files: test_generator.py
Objective: Create a program that automates part of writing unit tests
Write a program that produces a working skeleton python program to unittest a module.

This program should accept the module-under-test-name as a parameter, run without errors and fail all tests. You can print all program data to the console, and pipe it to a file like this: `./test_generator.py temperature >test_temperature.py`

So this excercise is not yet about reading or generating files!!

To be able to generate this program, you'll need to get a list of the functions in a module in some way. Here are a few suggestions, try these in interactive mode first :

```
>>> import temperature
>>> dir(temperature)
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
'c2f', 'c2k', 'f2c', 'f2k', 'k2c', 'k2f']
```

Another way could be:

```
>>> import temperature
>>> temperature.__dict__
{'f2c': <function f2c at 0x100495cf8>, 'k2c': <function k2c at
0x100490f50>, ... <a lot of other output> ...

>>> import types
>>> for k,v in temperature.__dict__.items():
...     if isinstance(v, types.FunctionType) : print k
...
f2c
k2c
f2k
c2f
c2k
```

```
k2f
```

The nicest and most pythonic way would be to use the `inspect and importlib` modules. We need the module object to use with `inspect` and we get it with `importlib`. This is how you dyamically load modules in Python.

Use this way of working, we'll build more on top of this later ...

```
>>> import importlib, inspect
>>> mod_to_test = importlib.import_module("temperature")
>>> inspect.getmembers(mod_to_test)
<lots of output, try it...>

>>> inspect.getmembers(mod_to_test, inspect.isfunction)
[('c2f', <function c2f at 0x10065c830>), ('c2k', <function c2k at 0x10065c170>), ('f2c', <function
```

3.3 C UnitTest Skeleton generator

Files: c_test_generator.py, test_dummy.c
Objective: Create a program that automates part of writing C unit tests with Check

This exercise is about writing a C UnitTest Skeleton generator.

We're not going to read or write any files...
Our program will accept the "module-under-test" as a program parameter, followed by the name of the functions to be tested.

You can print all program data to the console, and pipe it to a file like this:
`./c_test_generator.py dummy function another_function > test_some_c_module.c`

Write a module containing a function, that accepts a list of function names. Generate ready to use C skeleton code for a Check UnitTest of these functions. The generated UnitTest code should compile, run and fail all tests. You may reuse the makefile from the solutions zip.

To be clear, you don't have to "read C code" programmatically in this example, we'll do that later, and use this module to produce other C code from it.

Output for C module called `dummy`, with 2 functions `function` and `another_function` should look like:

```
//Automatically generated code to Check module  dummy

#include <stdlib.h>
#include <check.h>
#include "dummy.h"
//-------------------------------------------------------------
START_TEST (test_function)
{
    ck_assert(0);
}
END_TEST
//-------------------------------------------------------------
START_TEST (test_another_function)
{
    ck_assert(0);
}
END_TEST
//-------------------------------------------------------------
Suite* dummy_suite()
{
    Suite* s = suite_create("dummy");
    TCase* core = tcase_create("Core");
    tcase_add_test(core, test_function);
    tcase_add_test(core, test_another_function);
    suite_add_tcase(s, core);
    return s;
}
//-------------------------------------------------------------
int main()
{
    int number_failed;
    Suite *s = dummy_suite();
    SRunner *sr = srunner_create (s);
    srunner_run_all (sr, CK_NORMAL);
    number_failed = srunner_ntests_failed (sr);
    srunner_free (sr);
    return (number_failed == 0) ? EXIT_SUCCESS : EXIT_FAILURE;
}
//-------------------------------------------------------------
```

1 Strings & Files

1.1 Piglatin

Files: piglatin.py, test_piglatin.py
Objective: Gain more familiarity with string operations and files

1.1.1 To piglatin

Objective: Write piglatin code

Write a module `piglatin` containing a `to_piglatin()` function.

Piglatin is a language game where words in English are altered according to a simple set of rules.

The usual rules for changing standard English into Pig Latin are as follows:

For words that begin with consonant sounds, the initial consonant or consonant cluster is moved to the end of the word, and "ay" is added, as in the following examples:

- "happy" → "appyhay"
- "duck" → "uckday"
- "glove" → "oveglay"

For words that begin with vowel sounds or silent letter, you just add ay to the end. Examples are:

- "egg" → "eggay"
- "inbox" → "inboxay"
- "eight" → "eightay"

Generate a unit test using your generator, and complete the skeleton code with actual tests.

1.1.2 Piglatin program

Files: piglatin.py
Objective: Write a program around previous piglatin code, that reads from files

Make your `piglatin.py` also a program (only run program code when not used as module) that accepts 2 arguments, an input and output file. The program uses the module to convert the input file to piglatin and writes it to the output file.

1.1.3 From piglatin

Files: piglatin.py, test_piglatin.py
Objective: Implement the reverse piglatin code

Refactor your `to_piglatin()` implementation so a '-' is inserted before the part you moved to the end to disambiguate things. 'beast' would thus be 'east-bay'.

Write the `from_piglatin()` function + tests.

1.2 Refactor code generator (format)

Files: test_generator.py
Objective: Use more advanced text generation functionality

Refactor your Python code generator so that it uses `format()` template style string formatting to write to a file instead of printing each line individually.

If you didn't do this exercise before, you can always start from the one in the solutions in the modules directory. Alternatively you could also skip to the Wordcount exercise.

1.3 Refactor C UnitTest generator program (regex)

Files: c_test_generator.py
Objective: Use more advanced text generation functionality

Refactor the program in your C UnitTest generator so that it accepts a list of header files for which it will generate UnitTest skeletons.

If you didn't do this exercise before, you can always start from the one in the solutions in the modules directory. Alternatively you could also skip to the Wordcount exercise.

Create your regex based on the following assumption about function declarations:

```
one x characters in a to Z, followed by x spaces, followed by one
character in a to Z, followed by x characters in a to Z or 0 to 9,
followed by (, followed by x characters, followed by );


int max(int a, int b);   // only simple declarations for now
```

If you're new to regular expressions, ask the trainer for extra explanation or the actual regex. Don't loose too much time with your regex!

1.4 Wordcount

Files: wordcount.py
Objective: Train your file and dict skills with a Google exercise

In the "basics" directory of the Python exercises by Google, you'll find the `wordcount.py` files.

The synopsis for this program is `./wordcount.py {--count | --topcount} file`.

Based on the flag it wil either print how often each word appears in the file. Or give a top 20 of most occuring words.
Some code was written that calls functions you'll have to define. These are `print_words(filename)` and `print_top(filename)`. You can write helper functions to avoid duplicate code.

More requirements are to be found in `wordcount.py`, along with Googles advice not to code this all at once, but develop incrementally. This is good advice! Follow it!

1.5 Mimic

Optional exercise. See `mimic.py` for details.

2 Iterators

2.1 Traffic cameras

Files: query_cameras.py
Objective: Query data in text files
Write a program that queries a combination of text files for traffic cameras based on certain criteria.

2.1.1 The ground work

Objective: Parse lines of text data

In the solutions directory you'll find files from "Agentschap Wegen en Verkeer" containing positions of radar and loop based traffic cameras. Your program should be able to parse these files. Start with file "`snelheidscamera's_met_lus_120601.csv`".

Setup a basic argparse to accept a `--cameras` argument for the input file. This is a mandatory argument.

Start by writing a file-parser routine. It should accept anything that's iterable. Think about how to pass data from the file to the routine efficiently.

The first 2 lines in the camera files don't contain camera data and should be skipped. Think about how to efficiently skip those in the iterable. (Don't use slicing! Be more efficient)

The last 9 lines or so are also not camera data, so they can be removed as well.

Before continuing the exercise after writing this method, verify the output of the function is a tuple of tuples like: `('West-Vlaanderen', 'Vlamertinge', 'N38', '10.170', 'richting Poperinge')`. Also vefiry the number of parsed items.

Implement a few hardcoded filters of your choice. Like all cameras on E40 or in Gent... Use a combination of `map` and `filter` using `lambda`, instead of list comprehensions

This exercise will take a while, don't spend too much time on blocking issues, collaborate with your colleagues and trainer.

2.1.2 The program

Objective: Write a program to use the parser

Setup `argparse` to conform to the following:

`./query_cameras.py -h` should produce:

```
usage: query_cameras.py [-h] --cameras CAMERA_FILE [--cameras-in-city CITY]
                        [--cameras-on-road ROAD]

Queries input text files for Cameras, based on certain criteria

optional arguments:
  -h, --help            show this help message and exit
  --cameras CAMERA_FILE
                        A properly formatted file containing locations of
                        cameras
  --cameras-in-city CITY
                        Queries input files for all cameras in a city
  --cameras-on-road ROAD
                        Queries input files for all cameras on a certain road
```

The "cameras" is a required argument. `argparse` has support for required options. Running the program without parameters should print something like:

```
ewoudvc:iterators ewoud$ ./query_cameras.py
usage: query_cameras.py [-h] --cameras CAMERA_FILE
                        [--cameras_in_city CITY] [--cameras_on_road ROAD]
query_cameras.py: error: argument --cameras is required
```

For one or more of the non-required arguments, the program produces the result independantly.

2.1.3 More input files

Objective: Add support for multiple formats of data

Add support for a second input file from AGW to your parser:
The files from AGW have slightly different formats. The one file combines the road type and number in one field, like `E19`, or more confusingly also `E 40`, while the other file uses 2 fields, like `E,19`. The options shall accept the `E19` format. In the next subsection, you'll refactor your routines to detect and correct.

Another thing that's different is the notation of the kilometer mark for the camera. The one file has `38.28` for example, while the other has `"10,170"`. Notice the field separator in the quoted string. Defer this until the subsection.

2.1.4 Correct for boguous data

See the previous remarks about slight differences in the file formats. Refactor to detect and correct these.

2.2 Traffic cameras Extension

Files: query_cameras_extension.py
Objective: Support search by area code
Extend the previous program so that it also accepts an areacodes file and an option to query cameras in an area code.

Parsing the areacodes file should not present any problems. The first 2 lines are of no importance, skip them efficiently. Before continuing, vefiry the correct operation of this parser.

The argparse setup should be as follows:

```
ewoudvc:iterators ewoud$ ./query_cameras.py -h
usage: query_cameras.py [-h] --cameras CAMERA_FILE --areacodes AREACODES_FILES
                        [--cameras_in_areacode AREACODE]
                        [--cameras_in_city CITY] [--cameras_on_road ROAD]

Queries input text files for Cameras, based on certain criteria

optional arguments:
  -h, --help            show this help message and exit
  --cameras CAMERA_FILE
                        A properly formatted file containing locations of
                        cameras
  --areacodes AREACODES_FILES
                        A properly formatted file containing area codes for
                        cities
  --cameras_in_areacode AREACODE
                        Queries input files for all cameras in an area code
  --cameras_in_city CITY
                        Queries input files for all cameras in a city
  --cameras_on_road ROAD
                        Queries input files for all cameras on a certain road
```

Running the program with `--cameras-in-areacode=xxx` should produce something like:

```
ewoudvc:iterators ewoud$ ./query_cameras.py \
--cameras=snelheidscamera\'s_met_radar_120601.csv \
--areacodes=postcodes.csv --cameras-in-areacode=8400
Cameras in 8400 Oostende
Camera on R31 at km 2.550 t.h.v. Q8 tankstation - richting centrrum in Oostende (West-Vlaanderen)
Camera on R31 at km 1.390 t.h.v. Eendrachtstraat in Oostende (West-Vlaanderen)
Camera on N33 at km 3.397 richting Gistel in Oostende (West-Vlaanderen)
Camera on N33 at km 3.623 richting Centrum in Oostende (West-Vlaanderen)
Camera on N318 at km 2.300 richting luchthaven in Oostende (West-Vlaanderen)
```

3 Exceptions, assert and doctest

3.1 Temperature Conversion Module refactoring

Files: temperature.py, test_temperature.py
Objective: Get comfortable with Exceptions by adding error checking and corresponding tests

Copy and refactor your temperature conversion module so that it asserts that a temperature higher than the absolute zero is passed to any of the functions. Add `assertsRaises` to your unit test case for those values, checking if the `AssertionError` is raised.

Alternatively you could refactor the temperature module from the solutions in the modules directory.

3.2 Traffic cameras refactoring

Files: query_cameras.py, camera_parser.py
Objective: Raising an Exception when data sanity check fails

3.2.1 Raising exceptions

Copy and refactor your Traffic cameras program. Separate the parser functions into a module. Check each input line for the desired format and `raise` an exception if incorrect. Use `ValueError` or just `Exception` for example, we'll cover later how to create your own exception types.

If your Traffic camera's implementation isn't finished, you can also refactor the one from the solutions in the iterators directory.

3.2.2 Doctest

Create doctests from an interactive session where you tested good and bad case scenarios.

1 Classes

1.1 Classic shapes

Files: shapes.py Objective: Write your first Python class

The classic Oo shape example, but without inheritance (as Python doesn't need it to make this exercise work).

Define 2 Shape classes, Square and Circle, add methods `getSurfaceArea()` and `getType()` to both. Write a method `printSurfaceAreaOfObject()` outside the classes that accepts an object, and prints a message using the data from those 2 functions.

Both classes take the required data in their constructors. The length of side and the radius respectively.

Instanciate at least one object of both classes and call `printSurfaceAreaOfObject()` with the object as argument.

Use `math.pi`

1.2 Point class

Files: point.py
Objective: Write special methods for a class

Write a Point class containing an x and y variable. Have the init method accept these on creation. Implement `__str__` so it prints something like `Point(33,42)`. Add a move method accepting an x and y to move it by. And a method that takes another Point and calculates the distance between those in x and y.

```
p1 = Point(10,10)
print(p1)      # should print Point(10,10)
p2 = Point(0,0)
p1.distance(p2) # should print 10,10
```

1.3 Power outlets (basic)

Files: poweroutlet.py
Objective: Think about abstracting a piece of hardware

This exercise has an advanced twin in the next section. This basic exercise is part of the larger one. If you plan to do the larger one, skip this section and go straight there.

Imagine a testing application where you need to power cycle devices. You have a controllable power outlet device at your disposal of the imaginary brand "AwesomeOutlets". Such an outlet bank contains 8 individually controllable power outlets.

Talking to a outlet bank is done over any serial connection (uart, spi, usb, ethernet).

The protocol is as follows:

- `%1=0;` for setting output 1 off
- `%6=1;` for setting output 6 on
- Numbering starts at 1

Design a class that represents one of these individual outlets. It will need to keep an outlet id as a member variable. The class will recieve this id through the init method (constructor equivalent).

The class will also receive a (communication) object on which to write the protocol data. Assume this object has a write method, taking one argument, which could be a string.

The class has either one method taking a bool for manipulating it's state. Or a duo of on/off functions. These are the methods that write to the communication object.

1.4 Revisit earlier exercises

For non-Tass'ers: If object orientation isn't your thing, after the first exercises in this chapter, you can continue working on exercises from previous chapters

1.5 Exception class

Files: query_cameras.py, camera_parser.py

Objective: Create and use your own exception type
Refactor your "Traffic cameras" parsers to throw an exception type you created. Subclass it from `Exception`.

If your traffic cameras is not up to this, start from the solution's "Traffic cameras" in the exceptions directory.

1.6 Refactor Python UnitTest generator (classes)

Files: test_generator.py
Objective: Add support for testing classes

Refactor your Python UnitTest generator so it can generate unit tests for classes in a module. First generate a UnitTest class with methods for all (non-member) functions in the module, and then a UnitTest class per class in the module, with a test-member per member.

Take a look at the `inspect` module again, you'll need a different predicate to get the classes from the module.

Start from the solutions if your generator isn't up for refactoring.

1.7 Power outlets (advanced)

Files: poweroutlet.py, test_poweroutlet.py, awesome.py, test_awesome.py, youcontrols.py, test_youcontrols.py

Objective: Think about abstracting a piece of hardware, and layering to decoupling the creation and use of the abstraction

Imagine a testing application where you need to power cycle devices. You have controllable power outlets to your disposal of 2 different imaginary brands: AwesomeOutlets and YouControls. Both are controllable from some kind of serial interface, but use a different protocol. They both have multiple models available:

- AwesomeOutlets
  - AO-4: bank with 4 controllable power outlets
  - AO-8: bank with 8 controllable power outlets
  - AO-16: ...
- YouControls
  - YC6: 6 controllable power outlets
  - YC12: 12 controllable power outlets

1.7.1 poweroutlet module

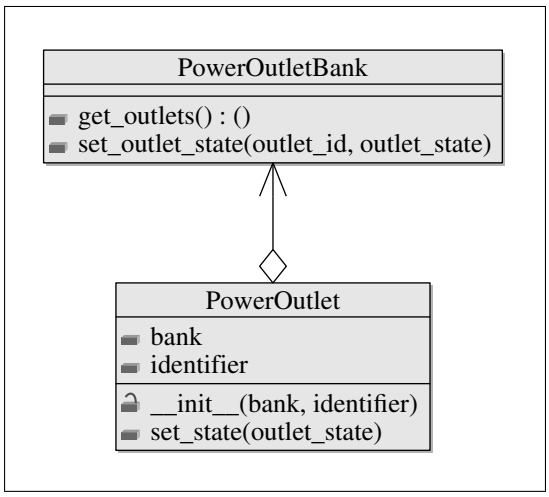
You as the Python expert need to create Python abstractions of "power outlet banks" to integrate with. Start by creating a module `poweroutlet` containing "base classes" `PowerOutletBank` and `PowerOutlet` conforming to the following:

```
>>> help(poweroutlet.PowerOutletBank)
Help on class PowerOutletBank in module poweroutlet:

class PowerOutletBank(builtins.object)
 |  "abstract base class" for any power outlet bank abstraction
 |
 |  Methods defined here:
 |
 |  get_outlets(self)
 |      Returns a tuple of PowerOutlet objects in this bank
 |
 |  set_outlet_state(self, outlet_id, outlet_state)
 |      Sets outlet correstponding to outlet_id to state corresponding outlet_state

>>> help(poweroutlet.PowerOutlet)
Help on class PowerOutlet in module poweroutlet:

class PowerOutlet(builtins.object)
 |  "base" for any power outlet abstraction
 |  Usable as a generic abstraction as is
```

```
|
|  Methods defined here:
|
|  set_state(self, outlet_state)
|      Sets the state of this outlet to the corresponding outlet_state
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  bank = None
|
|  identifier = None
```

> As a side note: Strictly speaking, in Python it's totally not needed to define this base class here, duck typing will make this work either way. But it can serve as a interface documentation of what compatible types should implement ... And as an exercise ...

We created the `PowerOutlet` class because test cases that need to control an outlet, don't need to know it might belong to a bank or whatever, they just want to use outlets. The outlet itself knows it belongs to a bank and will take care of that.



The `PowerOutletBank` methods should both raise a `NotImplementedError`. `PowerOutlet.set_state` should call `set_outlet_state()` on it's `bank` attribute, using it's `identifier` attribute.

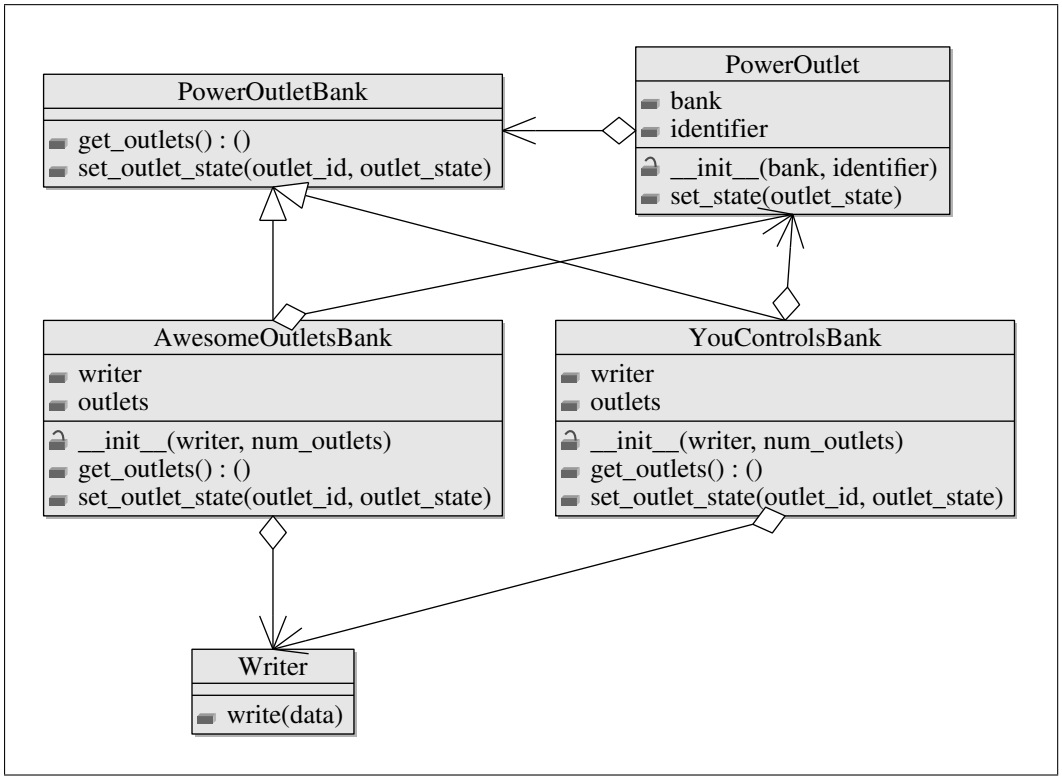
Define 2 "outlet states" in the base classe module. Like `PowerOn` and `PowerOff`.

Write a unit test module for these 2 classes. Also test your implementation against the unit tests in the solutions!! Don't forget coverage ...

1.7.2 Brand specific modules

Derive 2 classes `AwesomeOutletBank` and `YouControlsBank` from `PowerOutletBank` in their own respective modules and implement the methods. (Start with one manufacturer).

These classes will be where you implement the protocols. And since those protocols need to go somewhere, you'll need to "inject" an object (dependency) which supports writing (=has a method named write accepting something) into them. You do this as a "constructor" argument.



These classes should also take a count as a "constructor" argument. You could create convenience subclasses for each brand-model, like `AwesomeOutletAO4` for example which passes the correct count to it's parent. Up to you ...

The constructor should assert that the count makes sense for that perticular brand. It should also assert the write parameter has the necesary write method. See `hasattr()`.

`set_outlet_state` should also assert the `outlet_id` is in an acceptable range for that instance, and that the state is one of `PowerOn` or `PowerOff`.

The protocols are as follows:

- AwesomeOutlets
  - `%1=0;` for setting output 1 off
  - `%6=1;` for setting output 6 on

  - Numbering starts at 1

- YouControls
  - `000001\n` for setting output 0 on, the rest off
  - `000011\n` 0 and 1 on, the rest off

Start with one manufacturer, and try writing a unit test for this. How will you aproach this? What does your gut tell you you lack here? Run your code against the unit tests in the solutions. You could take a sneak peak to the implementation as well. Write your second implementation, and unit test against solutions again.

2 Filesystem

2.1 Course grading system

Files: coursegrading.py
Objective: Learn about dealing with archives and the filesystem

Imagine you're a course teacher, and you want to verify that course participants solved exercises according to certain criteria. It's not a real stretch of this imagination that a teacher is quite lazy but loves to program. So you're going to automate the screening of participant solutions.

In this exercise we're automating the following:

- Checking if an archive name follows a certain format
- Unpacking of said archive
- Checking if the directories for a certain chapter are present
  - You can limit yourself to a number of chapers of your choice
- Checking if for the present chapters, the expected files are also present

Just print a little report of this, in the line of :

```
Archive under test:  ../filesystem/Course-Teacher-PythonFundamentalsSolutions.zip
Is correctly named, by author: Course Teacher

Found exercise directories:
     classes
     data_types
     exceptions
     functions
     modules
     strings

Lacking exercise directories:
     iterators
     user_iterators
     filesystem

Found the following solutions in directory classes
     test_poweroutlet.py
     test_awesome.py
     test_youcontrols.py
     test_generator.py
     poweroutlet.py
     awesome.py
     youcontrols.py
```

```
Directory classes lacks the following:
     camera_parser.py
     query_cameras.py

... <and so on>
```

In following exercises we'll build other automation on top of this.

The format for the archive name is the following:
`FirstName-LastName-PythonFundamentalsSolutions.zip`

2.1.1 Directories

Start by checking the directories in the zip. Directories contain the name of the exercises chapter, in lowercase: like `filesystem` for this chapter. Check which are present, absent and stray.

2.1.2 Files

You should have list of files that needs to be present in each chapter. This way you can check, present, lacking and stray files.

3 Subprocesses

3.1 Course grading system

Files: coursegrading.py, examine.py, check_data_types.py, check_functions.py, check_modules.py ...
Objective: Launch subprocesses and output status

This exercise builds on the Course grading system exercise from the Filesystems chapter. If you haven't finished it, you can continue working on it, or use the solution from the solutions zip and start working from there.

In this exercise we're automating the following:

- Running programs in a participant solution dir
- Checking if the output of these programs conforms to the exercise assignments
  - Some must run with returncode 0
  - Some with errorcode 0 can still fail (see google exercises)
  - Others must fail with a specific message

Use the `subprocess.call()` and/or `subprocess.check_output()` functions we covered in the slides.

Just print to the screen if a certain exercise runs as expected. Here's how to check:

- data_types
  - string1.py string2.py list1.py list2.py
    - At least they should return error code 0, check this
    - But even with 0 success is not guaranteed
    - They print to the screen whether or not they succeeded
    - In the sources you can see that ' OK ' is printed for success
    - ' X ' for failure
    - As soon as one ' X ' is detected, the exercise is regarded as failing
- functions
  - number_tester.py dict_inverse.py transpose.py
    - Should run with error code 0
    - Anything else is regarded as failing
- modules
  - test_temperature.py
    - Samething as for functions
  - temperature.py
    - Should fail with non zero error code if presented with wrong command line arguments
    - Should succeed with 0 if no command line arguments, or correct ones

3.1.1 Report

Extend your previous report with the results of these checks. Do at least `base_types, functions and modules`. Write a helper module per chapter, `check_base_types, check_functions, ...`. The helper module should contain a list of files to be present, and a function `check` that runs the programs and does the checks, producing a dictionary, to be added to the report by the caller. Refactor your earlier implementation to support this.

Separate the reporting from the "business logic" of examining the zipfile. Put the logic in an `examine.py`.

For `modules`, you'll need to change the permission bits on the "check" executable. See `os.stat` and `os.chmod`.