Agenda

Python Basics

- description
- python scripts, python cmd
- basic language syntax
- python-specific conditional operators
- builtin types, builtin data structures and usage
- strings
- fstrings
- array slicing
- if, range, for
- list comprehension
- dictionaries
- imports, import as
- files
- packaging
- error handling

Python Functions

- syntax, default values, *args, **kvargs
- argument types
- yield statement (generator)
- context manager
- decorators
- lambdas

Python OOP

- class syntax
- encapsulation, polymorphism and inheritance
- abstract classes
- duck typing

Python Advanced

- class decorators
- metaclasses
- pdb
- multithreading

- multiprocessing
- pip, requirements
- linting, pep8
- cookiecutter
- tox
- unit tests, mocking

Exercises

Python Basics

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. It's focused on being simple, easy to learn and emphasizes readability. It can also be used as a scripting language.

It can be run directly in the terminal by running `python3`, or used to run python3 scripts by running `python3 script_name.py`.

Alternatively, on Linux the header `#!/usr/bin/env python3` can be added to a file, allowing it to be executed directly, after having execute permissions granted.

For Python projects, executable files can also be generated based on the project's entry points defined in its `setup.cfg` file.

Python doesn't explicitly need a main function, all the code in a python script will be executed. However, if that's not desired, a "main" function can be created:

```
def main():
    print("Hello world!")


# This guard statement is useful to prevent the code from executing when
# importing the module.
if __name__ == "__main__":
    main()
```

Easter eggs:

```
from __future__ import braces
import antigravity
import this
```

Basic Language Syntax

Basic types:

```
ubuntu@ubuntu:~$ python3
>>> print("Hello, I am a string!")
Hello, I am a string!
>>> foo = 4.2
>>> type(foo)
<class 'float'>

>>> arr = [4, 4, 4.5, None, True]
>>> print(arr)
[4, 4.5, None, True, [1, 2, 3]]
>>> print(set(arr))
{None, 4, 4.5, True}
>>> print({1, 2, 3, "I am a tuple"})
{1, 2, 3, 'I am a tuple'}
```

NOTE: All the elements of the `array` should be hashable in order to transform the `array` into a `set`. A `set` has a random order of elements and each element is **unique**. A `tuple` is an immutable array.

Strings:

```
>>> "some" + "thing"
'something'

>>> "thing" in "something"
True

>>> s = """Multi
... line
... string"""
>>> s
'Multi\nline\nstring'

>>> s.split("\n")
['Multi', 'line', 'string']
>>> " ".join(s.split("\n"))
'Multi line string'

>>> # String formatting.
>>> "Formatting %s in here." % "something"
'Formatting something in here.'
```

```
>>> "Formatting %s and %s in here." % ("something", "other")
'Formatting something and other in here.'

>>> "Formatting keywords in here, like %(name)s" % {"name": "Gigi"}
'Formatting keywords in here, like Gigi'

>>> # Returns all of the string properties / methods.
>>> dir(str)
>>> # Prints the help / docstring of a method.
>>> help(str.find)
>>> s.find("line")
6
```

Array slicing:

```
>>> # Syntax: array[start:end:step]
>>> arr[0:3]
[4, 4, 4.5]

>>> # Slice all the elements except the last one.
>>> arr[:-1]
[4, 4, 4.5, None]

>>> # Slice every other element.
>>> arr[::2]
[4, 4.5, True]

>>> # Having the step -1 walks the list in reverse.
>>> "esrever ni retteb kool dluow siht"[::-1]
'this would look better in reverse'
```

Exercise - Array slicing

R2-D2 wants to get back home, yet he encountered some problems. The text that will get him back home safely is in th
e wrong order. Who will be able to help R2-D2 get back home safely?

Task: Help R2-D2 get home by using array-slicing so that the final text will look like this: "R2-D2 va ajunge acasa"

```
text = ["acasa", "ajunge", "va", "nu", "R2-D2"]
```

If:

```
>>> d = {"some_key": "some_value", 2: 5, None: "foo", "other": {}}
>>> d
{'some_key': 'some_value', 2: 5, None: 'foo', 'other': {}}

>>> if 4 in arr:
...     print("The array has the element!")
...
The array has the element!

>>> if "some_key" in d.keys():
...     print("The dict has the 'some_key' key!")
...
The dict has the 'some_key' key!
```

range, for:

```
>>> help(range)
>>> # range(start, end, step) returns a generator, not an actual list.
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10, 2)
range(5, 10, 2)

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

Exercise - If, range, for

Anakin and Obi-Wan are fighting over the number of vowels and consonants of the expression 'hodoronc-tronc'. Is there a chance of this fight to get to an end?

Task: Your task is to write a python script that will calculate the number of vowels and consonants of the expression 'hodoronc-tronc' using `if`, `range`, `for` learnt previously.

```
text = "hodoronc-tronc"
vowels = "aeiou"
```

List comprehension:

```
>>> # Creating a new array:
>>> arr = [10, 20, 3, 6, 42]
>>> new_arr = []
>>> for element in arr:
...     if element % 2:
...             new_arr.append(element ** 2)
...
>>> new_arr
[9]

>>> # List comprehension
>>> new_arr = [x ** 2 for x in arr if x % 2]
>>> new_arr
[9]
```

Exercise - List comprehension

Jabba just started learning python and he discovered something called `list comprehension`. He wishes to create a list, which will contain all the even numbers from `[1, 2, 3, 4, 5, 6, 2, 8, 6, 5, 3, 1, 9]` using this new technique that he just learnt. How should he do this?

Task: Write a python script that generates a list containing all the even numbers only once from Jabba's list using list comprehension.

Hint: Jabba's best friend is called Set.

```
Jabba_sir = [1, 2, 3, 4, 5, 6, 2, 8, 6, 5, 3, 1, 9]
```

Dictionaries

```
# Define dictionaries
my_dict = {'name': 'Obi-Wan Kenobi', 'age': 48}
other_dict = dict(name="Obi-Wan", age=48)

print(other_dict)
# Output: {'name': 'Obi-Wan', 'age': 48}

print(my_dict.get('age'))
# Output: 48

print(my_dict.keys())
# Output: dict_keys(['name', 'age'])
```

```
print(my_dict.values())
# Output: dict_values(['Obi-Wan Kenobi', 48])

my_dict['age'] = 50
print(my_dict)
# Output: {'name': 'Obi-Wan Kenobi', 'age': 50}

my_dict['Homeworld'] = 'Stewjon'
print(my_dict)
# Output: {'name': 'Obi-Wan Kenobi', 'age': 50, 'Homeworld': 'Stewjon'}

del my_dict['age']
print(my_dict)
# Output: {'name': 'Obi-Wan Kenobi', 'Homeworld': 'Stewjon'}

# Iterate over a dictionary
for key in my_dict:
    print(key, '->', my_dict[key])

# Output:
# name -> Obi-Wan Kenobi
# Homeworld -> Stewjon
```

Imports, import as

```
import math

import numpy as np

from math import sqrt, pi
```

Python allows you to import all the names from a module (e.g.: `from math import *`). It's tempting to use this instead of prefixing those names with `math.` throughout your code, but please don't! Wildcard imports can cause more name collisions because you can't see the explicit names being imported, and it makes problems hard to debug. It also becomes difficult to say which module provides a given type. So stick to explicit imports!

```
# built-in json parsing library
import json

# Very useful for parsing yaml
import yaml

# Useful for parsing TOML documents
import toml
```

Files

```
import json  # We will use a json

dic = {'name': 'John', 'age': 30}

# Open the file to write
file = open("data.json", "w")

# Write the dictionary inside the json file
json.dump(dic, file)

# Close file
file.close()

# Open the file to read
file = open("data.json", "r")

# Read the dictionary from the json file
new_dic = json.load(file)
print(new_dic)
# Output: {'name': 'John', 'age': 30}

# Close file
file.close()
```

Packages:

Python code can be structured in packages (folders) and modules (python files). For example, if we have the following files in the current folder:

```
foo/bar.py
foo/lish.py
```

Then, the modules `bar` and `lish` are in the `foo` package, and can be imported as follows:

```
from foo import bar, lish
```

Note that these modules were imported relative to the current path. If a module is not relative to our path, it will be searched for in a few default locations, among Python's native libraries or installed 3rd party libraries, typically installed using `pip install`. The list of locations can be expanded through the `PYTHONPATH`

environment variable, which can be a string containing a list of directories separated by `:` on Linux or `;` on Windows. The `PYTHONPATH` environment variable can be useful in cases in which you developed a few Python modules but haven't made them installable yet (by using `setuptools`), but you'd still want to use them. Let's see where a few of these modules reside in:

```
import json
import pep8
import flake8
import requests

print(json.__file__)
# '/usr/lib/python3.8/json/__init__.py'

print(pep8.__file__)
# '/usr/local/lib/python3.8/dist-packages/pep8.py'

print(flake8.__file__)
# '/usr/local/lib/python3.8/dist-packages/flake8/__init__.py'

print(requests.__file__)
# '/home/ubuntu/.local/lib/python3.8/site-packages/requests/__init__.py'
```

Additionally, aliases can be given to the modules, which is especially useful for modules with conflicting names:

```
from foo import lish as foolish
```

A package may have an `__init__.py` file, representing the package itself, which will always be executed first whenever the packages is accessed. This file can be used to initialize modules (it executes only once, no matter how many times it is imported), or expose only certain attributes of the package:

```
foo/__init__.py

# importing classes is usually discouraged.
from foo.bar import B
from foo.lish import C
```

Which can then be used as:

```
import foo

b = foo.B()
c = foo.C()
```

For unit tests, the `__init__.py` file still needs to exist in the test modules and submodules in order to be discoverable.

Importing packages will result in `__pycache__` folders being generated, which contains compiled Python code, meaning that the modules will not have to be compiled again on subsequent runs. The modules are recompiled automatically if the module's code changes.

```
ls foo/__pycache__/
bar.cpython-38.pyc  __init__.cpython-38.pyc  lish.cpython-38.pyc
```

Exercise - Dictionaries, files and imports

Leia just received a very hard task, which she must solve in 5 minutes. She must parse a JSON file and extract some information. Can you help her do this?

Task: You get a JSON file `data.json` containing different data about employees from a company. You have to parse the JSON and print the names of the employees as well as the department they work in like this: `Name: John Doe | Department: MMORPG Farming`.

```
{
  "employees": [
    {
      "id": 1,
      "name": "Han Solo",
      "department": "Engineering"
    },
    {
      "id": 2,
      "name": "Padme Amidala",
      "department": "Marketing"
    },
    {
      "id": 3,
      "name": "Bobba Fett",
      "department": "Finance"
    }
  ]
}
```

Error handling:

One or multiple exception types can be handled at once:

```
try:
    with open("foo.txt", "r") as f:
        print(f.read())
except FileNotFoundError as ex:
    print("Exception while opening file: %s" % ex)
except Exception as ex:
    print("Generic Exception: %s" % ex)
```

Python provides a keyword `finally`, which is always executed after the try and except blocks. The finally block always executes after normal termination of try block or after try block terminates due to some exception. Even if you return in the except block, the `finally` block will still execute.

```
try:
    with open("foo.txt", "r") as f:
        print(f.read())
except (FileNotFoundError, Exception) as ex:
    print("Exception while opening file: %s" % ex)
finally:
    print("This will **always** be executed!")
```

Python Functions

```
def is_palindrome(string):
    """Returns True if the given string is a palindrome, False
otherwise."""
    return string == string[::-1]


string = "racecar"
print("Is the word %s a palindrome: %s" % (string,
is_palindrome(string)))
```

Arguments with default values:

```
def function(foo, bar=None):
    print("Optional argument is: %s" % bar)


>>> function(5)
Optional argument is: None
>>> function(5, 10)
Optional argument is: 10
```

NOTE: Be careful what default argument values you set, there might be unintended side effects.

```
def function(foo, bar=[]):
    bar.append(1)
    print("Optional argument is: %s" % bar)


>>> function(3, [1])
Optional argument is: [1, 1]
>>> function(3, [1])
Optional argument is: [1, 1]
>>> function(3)
Optional argument is: [1]
>>> function(3)
Optional argument is: [1, 1]
>>> function(3)
Optional argument is: [1, 1, 1]
```

Correct way to handle this scenario:

```
def function(foo, bar=None):
    bar = bar or []
    bar.append(1)
    print("Optional argument is: %s" % bar)
```

Correct and incorrect ways to use arguments (including variable-length arguments `*args` (list) and `**kwargs` (named arguments)):

```
def function(foo, bar=2, *args, **kwargs):
    print("Arguments are: %s, %s." % (foo, bar))
    print("Variable arguments: %s" % (args, ))
    print("Key-value arguments: %s" % (kwargs, ))


>>> function(1)
Arguments are: 1, 2.
Variable arguments: ()
Key-value arguments: {}

>>> function(1, 5, 7, 8, 9, 10)
Arguments are: 1, 5.
Variable arguments: (7, 8, 9, 10)
Key-value arguments: {}

>>> function(1, 5, 7, 8, 9, other=10)
Arguments are: 1, 5.
Variable arguments: (7, 8, 9)
Key-value arguments: {'other': 10}

>>> function(foo=1, bar=5, other=3)
Arguments are: 1, 5.
Variable arguments: ()
Key-value arguments: {'other': 3}

>>> function(foo=1, 2, 5)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument

>>> function(1, 5, bar=6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'bar'
```

Argument types:

```
def is_palindrome(s: str) -> bool:
    return s == s[::-1]


>>> print(is_palindrome("racecar"))
True
>>>
>>> # Types are not enforced, this still works.
>>> print(is_palindrome([1, 2, 2, 1]))
True


# Type aliases
from typing import List
Vector = List[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]


>>> # passes type checking; a list of floats qualifies as a Vector.
>>> new_vector = scale(2.0, [1.0, -4.2, 5.4])
>>>
>>> new_vector
[2.0, -8.4, 10.8]

# Callables.
from typing import Callable

def process_items(get_next_item: Callable[[], str]) -> None:
    # Body
```

Exercise: Create a function `copy_fields` that returns a new dictionary that will only have the wanted fields from a given dictionary. The function will have the following parameters: the dictionary to be copied from, an optional default value if the field cannot be found in the dictionary, and a variable list of arguments representing the fields to copy.

yield statement:

Generators are a special type of iterables which can be looped over. On each iteration, the generator `yield`s a new element which may have been just created. This can be especially useful in cases in which you would have an enormous collection of data but which doesn't need to / can't be loaded entirely in memory and you're searching for a specific entry.

Examples of generators: `range()`, `dict.items()`, `dict.keys()`, `dict.values()`, list comprehension generators (`(x for x in array)`).

Example of a list comprehension generator:

```
>>> square_gen = (x ** 2 for x in range(10))
>>> square_gen
<generator object <genexpr> at 0x7fc48a825970>
>>> list(square_gen)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
```

Example of a generator function:

```
def fibonacci_generator():
    a, b = 0, 1
    while(1):
        yield a
        a, b = b, a + b


for n in fibonacci_generator():
    print(n)
    if n > 30:
        break


import itertools

slice = itertools.islice(fibonacci_generator(), 5)
list(slice)
[0, 1, 1, 2, 3]
```

Context Manager:

Typically, when working with certain resources, some cleanup may be required. For example, if you open a file, you should close the file after you're done with it. This can be done through a `__runtime context__` by using the `with` statement, which will call the context manager's `__enter__` function (setup code) when creating the context and `__exit__` function (teardown code) when the context ends. For example, the `open` function will open the file in its `__enter__` function, and closes it on its `__exit__` function. Note that `__exit__` will be called even in case of exceptions, receiving it as a parameter.

Example:

```
with open("sobolan.txt", "r") as f:
    print(f.read())
```

Equivalent:

```
from contextlib import contextmanager


@contextmanager
def fopen_wrapper(filename, mode):
    print(f"Opening file {filename} with mode {mode}...")
    f = open(filename, mode)

    yield f

    print("Finished processing file...")
    f.close()


>>> with fopen_wrapper("foo.txt", "w") as f:
...     f.write("Hello!")
...
Opening file...
6
Closing file...
```

Decorators:

A decorator / wrapper is a function that receives a function as a parameter and returns another function. The given function is typically "decorated" with additional functionality by adding preexecution and postexecution steps to the call itself. In some scenarios, the call itself is omitted. The ideal decorators are simple to understand, allowing the function themselves to be simpler as well.
Example:

```
def logger(func):
    def inner(*args, **kvargs):
        print("Args: %s, kvargs: %s", % (args, kvargs))
        result = func(*args, **kvargs)
        print("Result: %s" % result)

    return inner
```

```
@logger
def adding_func(a, b):
    return a + b


>>> adding_func()
```

Exercise: Create an Exception catcher decorator, which will catch all the exceptions, log them, and return `None` instead of having the exception be uncaught.

```
def error_wrapper(func):
    def inner(*args, **kvargs):
        try:
            return func(*args, **kvargs)
        except Exception as ex:
            print("Caught exception: %s" % ex)
            return None
    return inner


@error_wrapper
def failing_func():
    raise Exception("Expected exception")


>>> failing_func()
Caught exception: Expected exception
```

Decorators with parameters:

```
def error_wrapper(*exc_types):
    exc_types = exc_types or [Exception]
    def decorator(func):
        def inner(*args, **kvargs):
            try:
                return func(*args, **kvargs)
            except exc_types as ex:
                print("Caught exception: %s" % ex)
                return None
        return inner
    return decorator


@error_wrapper(FileNotFoundError)
def failing_func():
    f = open("file_that_doesnt_exist.txt", "r")
    f.close()


>>> failing_func()
```

Some example of Python decorators are:

- the `@contextmanager` decorator. Allows functions to be used in a context manager (`with`), allowing them to `yield` results and execute code after the context exists (e.g.: `open` is a function that has a context manager, and the file is automatically closed after the context ends).
- Flask's `@app.route` decorators.

Lambdas:

Lambdas are small anonymous functions, typically used in functional programming.

```
>>> foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]
>>> list(filter(lambda x: x % 3 == 0, foo))
[18, 9, 24, 12, 27]

>>> list(map(lambda x: x * 2 + 10, foo))
[14, 46, 28, 54, 44, 58, 26, 34, 64]

>>> import functools
>>> functools.reduce(lambda x, y: x + y, foo)
139
```

Python OOP

```
class Real:
    def __init__(self, a):
        self._a = a


class Complex(Real):
    def __init__(self, a, b):
        super().__init__(a)
        self._b = b


>>> c1 = Complex(1, 2)
>>> print("Complex number: %d * i + %d" % (c1._a, c1._b))
Complex number: 1 * i + 2
>>> print(c1)
<__main__.Complex object at 0x7f58c97c3f40>
```

Implementing the addition and string representation magic methods:

```
class Complex(Real):
    def __str__(self):
        return "%d * i + %d" % (self._a, self._b)

    def __add__(self, other):
        return Complex(self._a + other._a, self._b + other._b)

>>> c1 = Complex(1, 2)
>>> c2 = Complex(-3, 5)
>>> c3 = c1 + c2
>>> print("Complex number: %s" % c3)
Complex number: -2 * i + 7
```

Other magic methods:

- operator magic methods: `__sub__`, `__mul__`, `__floordiv__`, `__truediv__`, `__mod__`, `__pow__`, `__lt__`, `__le__`, `__eq__`, `__ne__`.
- construction magic methods:
  - `__new__(cls)`: called when a new object is created.
  - `__init__(self)`: called by the `__new__` method.
  - `__del__(self)`: destructor method.

- attribute magic methods:
  - `__getattribute__(self, name)`: called when accessing an attribute.
  - `__getattr__(self, name)`: called when accessing an attribute that does not exist.
  - `__setattr__(self, name, value)`: called when assigning a value to the attribute.
  - `__delattr__(self, name)`: called when deleting an attribute.

Overriding the `__getattribute__` magic method:

```
class Complex(Real):
    def __getattribute__(self, name):
        if name.startswith("_"):
            raise AttributeError
        return super().__getattribute__(name)

    @property
    def a(self):
        return super().__getattribute__("_a")

    @property
    def b(self):
        return super().__getattribute__("_b")


>>> from complex import Complex
>>>
>>> c1 = Complex(3, 4)
>>> c1.a
3
>>> c1._b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/ubuntu/workdir/temp/complex.py", line 13, in
__getattribute__
    raise AttributeError
AttributeError
```

Class properties, class methods, static methods:

```
class Complex(Real):

    # Class property.
    _count = 0
```

```
    def __init__(self, a, b):
        super().__init__(a)
        self._b = b
        Complex._count += 1

    @classmethod
    def get_count(cls):
        return cls._count

    @staticmethod
    def static():
        print("Just a static method.")


>>> from complex import Complex
>>> Complex.get_count()
0
>>> c1 = Complex(3, 4)
>>> Complex.get_count()
1
>>> c1.get_count()
1
>>> Complex.static()
Just a static method.
```

Abstract Classes:

An abstract class can be considered as a blueprint for other classes. It allows you to define a set of methods that must be implemented by any child classes inheriting the abstract class. A class which contains one or more abstract methods is called an abstract class. The module we can use to create an abstract class in Python is `abc` (abstract base class). An abstract method is a declared method without an implementation. To define an abstract method we use the `@abstractmethod` decorator of the `abc` module.

```
import abc


class Shape(abc.ABC):
    def __init__(self, shape_name):
        self.shape_name = shape_name

    @abc.abstractmethod
    def draw(self):
        pass
```

```
class Circle(Shape):
    def __init__(self):
        super().__init__("circle")

    def draw(self):
        print("Drawing a Circle")


# Create a Circle object:
circle = Circle()
circle.draw()
# Output: Drawing a Circle

# Instantiating a Shape will result in a TypeError:
shape = Shape()
```

Duck typing:

"If it looks like a duck, it walks like a duck, and quacks like a duck, it's probably a duck."

Types are not enforced, but the presence of attributes is:

```
class Ducky:
    def lay_eggs(self):
        print("%s laid an egg!" % self.__class__.__name__)


class Platypus:
    def lay_eggs(self):
        print("%s laid an egg!" % self.__class__.__name__)


>>> animals = [Ducky(), Platypus()]
>>> for animal in animals:
...     animal.lay_eggs()
...
Ducky laid an egg!
Platypus laid an egg!
```

Python Advanced

Class decorators

```
def exc_wrapper(func):
    def wrapper(*args, **kwargs):
        try:
            return func(*args, **kwargs)
        except Exception as ex:
            print("Exception caught: %s" % ex)
    return wrapper

def class_exc_wrapper(cls):
    callables = (
        (method_name, method)
        for method_name, method in cls.__dict__.items()
        if callable(method)
    )
    for method_name, method in callables:
        setattr(cls, method_name, exc_wrapper(method))

    return cls


@class_exc_wrapper
class A:
    def foo(self):
        raise Exception("Foo exception.")

    def bar(self):
        raise Exception("Bar exception.")


class B(A):
    def foo(self):
        raise Exception("Other foo exception.")

    def bar(self):
        super().bar()
        raise Exception("Other bar exception.")
```

```
>>> import sample
>>> a = sample.A()
>>> a.foo()
Exception caught: Foo exception.
>>> b = sample.B()
>>> b.foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sample.py", line 33, in foo
    raise Exception("Other foo exception.")
Exception: Other foo exception.
>>> b.bar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sample.py", line 38, in foo
    raise Exception("Other foo exception.")
```

Metaclasses

Metaprogramming refers to the potential for a program to have the knowledge and ability to manipulate itself. Metaclasses is a form of metaprogramming supported by Python.

Here's the quote from Tim Peter who wrote the Zen of Python:

`Metaclasses are deeper magic that 99% of users should never worry about it. If you wonder whether you need them, you don't (the people who actually need them to know with certainty that they need them and don't need an explanation about why).`

Everything is an object, including classes:

```
>>> class A:
...     pass
...
>>> type(A)
<class 'type'>
```

All classes are of type `type`. The call `A()` will invoke `A`'s `__call__` method, which will then call the methods `__new__` and `__init__`.

We can define a subclass of `type`, overriding these methods:

```
class ExcWrapperMeta(type):
    def __new__(cls, name, bases, dct):
        clazz = super().__new__(cls, name, bases, dct)
        clazz = class_exc_wrapper(clazz)

        print("Metaclass __new__ called")

        return clazz
```

We can then declare classes using this metaclass. Any new subclasses will inherit the metaclass as well:

```
class A(object, metaclass=ExcWrapperMeta):
    def foo(self):
        raise Exception("Foo exception.")

    def bar(self):
        raise Exception("Bar exception.")


class B(A):
    def foo(self):
        raise Exception("Other foo exception.")

    def bar(self):
        super().bar()
        raise Exception("Other bar exception.")


>>> import sample
Metaclass __new__ called
Metaclass __new__ called
>>> a = sample.A()
>>> a.foo()
Exception caught: Foo exception.
>>> b = sample.B()
>>> b.bar()
Exception caught: Bar exception.
Exception caught: Other bar exception.
```

Pdb

There is no perfect code, and we can't always expect it to work as magic. When something doesn't behave as expected, it's not very efficient to simply guess why it doesn't. There are multiple ways of knowing what is happening in the code, such as logging and debugging.

Learning to use a debugger is a useful endeavor, it's always useful for learning a new language as it can help you analyze certain features and different approaches, allowing you to understand something more deeply, or it can help you find the root of the problem (root cause analysis). The Python debugger in particular is also useful in development, giving you full control once a breakpoint has been reached, as if you're in a Python console.

To start debugging, you can simply add the following code anywhere in code:

```
import pdb; pdb.set_trace()
```

The code above will import `pdb` and set a breakpoint at that particular line. When that line is reached, we're given control over the execution in a Python terminal in which we have almost the same liberties as in any terminal: we can print variables, set variables, import new modules, call functions, etc. In addition to that, we also have `pdb`-specific commands that can help us debug:

- `help`: Lists all the debugger-specific commands. To note that most commands have a shortcut as well, typically the first letter of the command (`h` in this case).
- `help command_name`: Prints more details about that particular command.
- `list`: Prints a few lines above and below the current execution line. On new calls, it will continue listing the file below until `EOF`.
- `ll`: Always prints a few lines above and below the current execution line.
- `source function`: Prints the code for the given module, class, method, function, traceback, frame, or code object.
- `where`: Prints the current call stack, including files and line numbers.
- `args`: Prints the current function's arguments.
- `break location`: Sets a new breakpoint at the given `location`, which can be: a line number in the current file, or a function name, or a source filename and line number (`filename:10` - sets a breakpoint in `filename.py` at line 10).
- `clear breakpoint_number`: Removes the given breakpoint.
- `next`: Executes the current lines, goes to the **next** one.
- `step`: **Step** into the function call at the current line.

- `continue`: **Continue** until the next breakpoint.

In addition to that, there are a few other Python commandline commands that are useful when debugging, such as:

- `help(some_function_or_method)`: This will print the function's signature and docstring, which typically contains the documentation for that function, including potential argument, acceptable values, exception raised, etc.
- `dir(thing)`: Will return an array of properties associated with the given object. This can be very useful when you're not familiar with that particular object and its interface. It's very useful in tandem with `help`.

The `remote_pdb` is extremely useful for cases in which we do not have an interactive console available (e.g.: running as a service), allowing us to connect to te debugger remotely through a TCP port.

Multithreading

When starting a program, a process is being spawned, which will run the `main` function of the program. You typically only have one thread of execution (the `main` thread) which executes the code sequentially. However, there are many instances in which the thread will wait for certain operations to complete (I/O operations, file operations, syscalls, etc.), time which could be spent doing other needed operations. This issue could be solved through multithreading, allowing multiple threads to run in parallel, which could lead to reduced execution time. It is worth mentioning that the threads are not truly parallel. A process can only be executed by a single core, independently of how many CPUs a node has, which means that there can be only one thread running at a time. This is true even for processors with hyperthreading, which allows 2 threads to run "simultaneously" on the CPU, running one of them while the other one is waiting on I/O, cache miss, etc.

When using Python, you should be aware of its multithreading limitations. The CPython interpreter, which happens to be the most popular implementation, is subject to the Global Interpreter Lock (GIL), which prevents multiple threads from running Python code in parallel. The GIL is automatically released when performing I/O operations. Threads are also asked to release the lock after a certain amount of time has passed (5ms by default). As such, Python threads can run concurrently but not in parallel.

The situation is not as bad as it may seem. Python is commonly used for web applications that perform heavy IO, without significant computation. In this case, the GIL is released, allowing the IO operations to be performed in parallel.

Python also supports C extensions, which are allowed to release the GIL. Those can be imported like regular Python modules, but are actually implemented in C or C++. This allows offloading certain CPU intensive operations. Python also allows using C libraries (.so, .dylib, .dll) through `cffi` or `ctypes`.

Another way of achieving parallelism is through the use of separate processes, described below. Also worth mentioning that GIL is a CPython implementation detail, other implementations such as IronPython do not have it. There are in fact proposals to make it optional or even remove it altogether eventually.

```
import threading

def do_thing(the_thing):
    thread_name = threading.current_thread().name
    print(f"You know what the thing is? It is {the_thing}, "
          f"shown by {thread_name}.")

# We create a thread and specift the function to execute.
t1 = threading.Thread(target=do_thing, args=("Sobolan", ))
t2 = threading.Thread(target=do_thing, args=("Ezoteric", ))

# Start the threads.
t1.start()
t2.start()

# Wait for the threads to finish, otherwise we'd exit the program before
they
# would have a chance to run.
t1.join()
t2.join()

print("Finished execution!")
```

A better way to do this would be by using a Thread Pool, which is a set of threads created in advance that can be reused to execute multiple tasks:

```
from concurrent import futures


def work():
    print("Heyo!")


# Create a thread pool.
pool = futures.ThreadPoolExecutor(max_workers=3)

# make the thread pool do some work.
for i in range(10):
    pool.submit(work)

# Wait for all the work to finish.
pool.shutdown(wait=True)

print("Finished execution!")
```

Keep in mind that not all threads / tasks are independent of each other. For example, multiple threads trying to modify a unique resource can result in race conditions (the

threads are racing each other to it) in which the outcome is unpredictable and undesirable.

Let's consider the following code:

```
from concurrent import futures
import time

# Global resource / variable.
x = 0


def work():
    global x
    for i in range(100000):
        x = x + 1


def do_some_races():
    # reset the global x value.
    global x
    x = 0
    pool = futures.ThreadPoolExecutor(max_workers=3)
    for i in range(10):
        pool.submit(work)
    pool.shutdown(wait=True)


for i in range(10):
    now = time.time()
    do_some_races()
    elapsed = time.time() - now
    print(f"Attempt {i}: x = {x} in {elapsed} seconds.")
```

This issue can be solved by using a lock, allowing only the thread that has the lock to execute:

```
# Global resource / variable.
x = 0

# Lock for x.
lock = threading.Lock()

def increment_x():
    global x
    lock.acquire()
    x = x + 1
    lock.release()


def work():
    for i in range(100000):
        increment_x()
```

NOTE: the `Lock` has a context manager, which can be used with a context manager (`with`), which will acquire the lock when entering the context and releasing it on exit (including cases in which exceptions occur):

```
def increment_x():
    global x
    with lock:
        x = x + 1
```

We can even create a `@synchronized` decorator:

```
import functools
import threading

# Global resource / variable.
x = 0

# Lock for x.
lock = threading.Lock()

def syncronized(lock):
    def func(wrapped):
        @functools.wraps(wrapped)
        def inner(*args, **kwargs):
            with lock:
                return wrapped(*args, **kwargs)
        return inner
    return func
```

```
@syncronized(lock)
def increment_x():
    global x
    x += 1
```

NOTE: If threads requires multiple locks which can be aquired by other threads, there is a chance for deadlocks / livelocks, in which Thread A waits for Lock B and Thread B waits for Lock A.

Keep in mind that these threads and locks apply to only one process, each process having its own threads and locks. Synchronizing operations across multiple processes is a bit more difficult, especially if the processes are not even on the same node. In these cases, it's common practice to use file locks (they are just some simple empty files):

- a thread checks if the file lock exists in a particular location.
- if it doesn't exist, the thread creates it, effectively "acquiring" the lock.
- the thread deletes the file once it finishes its task, effectively "releasing" the lock.
- if the file already exists, it defers its execution until the file no longer exists.

One potential issue with this approach is that a thread / process could die before releasing the lock, blocking the other threads from ever acquiring the lock. Special measures should be taken to prevent this.

As an alternative to file locks, 3rd party software could be used as Distributed Lock Managers. Some popular services that can be used as Distribute Lock Managers are: Redis, etcd, zookeeper and even MySQL.

NOTE: It's quite hard to debug race conditions with a debugger. In this case, logging may be useful to at least notice them.

An important thing to note is that you can't access any data returned by a Thread's target. If needed, a mutable object like a dictionary could be passed as the function's arguments, which would allow the thread to set the result into the dictionary:

```
from concurrent import futures
import random


def work(data, index):
    print("Heyo!")
    data[index] = random.random()
```

```
# Create a thread pool.
pool = futures.ThreadPoolExecutor(max_workers=3)

# make the thread pool do some work.
data = {}
for i in range(10):
    pool.submit(work, data, i)

# Wait for all the work to finish.
pool.shutdown(wait=True)

print("Finished execution!")
print(list(data.values()))
```

A better way to do this would be to use a `Queue` (FIFO), which is also inherently thread-safe.

```
from concurrent import futures
import queue


def work(q):
    while True:
        thing = q.get()
        print(thing)
        q.task_done()


q = queue.Queue()
pool = futures.ThreadPoolExecutor(max_workers=3)

for i in range(3):
    pool.submit(work, q)

for i in range(100):
    q.put(i)

q.join()
pool.shutdown(wait=False)
print("Finished execution!")
```

Multiprocessing

Multiprocessing is another way to parallelize an application, allowing parts of it to run simultaneously. In some regards, it is quite similar to how multiple threads are being spawned and used:

```
import multiprocessing


def do_thing(the_thing):
    process_name = multiprocessing.current_process().name
    print(f"You know what the thing is? It is {the_thing}, "
          f"shown by {process_name}.")


# We create a thread and specify the function to execute.
p1 = multiprocessing.Process(target=do_thing, args=("Sobolan", ))
p2 = multiprocessing.Process(target=do_thing, args=("Ezoteric", ))

# Start the processes.
p1.start()
p2.start()

# Wait for the processes to finish, otherwise we'd exit the program
before they
# would have a chance to run.
p1.join()
p2.join()

print("Finished execution!")
```

A similar approach exists with `ProcessPoolExecutor`.

Discounting the similarities between the approaches above, there are quite a few differences between multiprocessing and multithreading. First of all, multiprocessing has true parallelism by having multiple processes to run on multiple different CPU cores (if there are more than 1), each of which having their own memory space, its own instance of Python interpreter (bypassing the GIL problem), and even its own threads. Because it's running on multiple CPUs, the performance can be much higher when compared to simple multithreading, especially since it won't have to do context switching between threads. However, there are some caveats as well:

- Because each process has its own memory space, typical variables and changes to them are localized to the process modifying them (unlike threads which share the same memory within the same process). There are however different interprocess communication (IPC) methods available (e.g.: `multiprocessing.Pipe`), but this may come with a performance penalty. An example would be shared memory between processes, such as `multiprocessing.Queue`, `multiprocessing.Array`, `multiprocessing.Value`, and `multiprocessing.Lock`:

```
from concurrent import futures
import multiprocessing


def work(q):
    while not q.empty():
        thing = q.get()
        print(thing)
        # If we received None, it means that we should stop.
        if thing is None:
            break

    print("Done!", flush=True)


if __name__ == "__main__":
    q = multiprocessing.Queue()
    pool = futures.ProcessPoolExecutor(max_workers=3)

    # Give some work to the processes.
    for i in range(3):
        pool.submit(work, q)

    # Put some items in the queue to be consumed.
    for i in range(100):
        q.put(i)

    # Stop the tasks.
    for i in range(3):
        q.put(None)

    # Wait for all the processes to finish working.
    pool.shutdown(wait=True)
    print("Finished execution!")
```

- The memory is being duplicated when creating a new process by forking (which isn't the case with threads), which can take time. In addition to this, there are some instances in which `multiprocessing` would fail simply because something was not serializable / picklable, especially on Windows, typically open resources which should only be open once, like files, sockets, etc. In order to avoid issues like this, multiprocessing should be used early and make sure that each process uniquely uses such resources.

- Child processes have their own `stdin`, `stdout`, and `stderr`, which could be handled in the Parent process. Because the child processes do not have the same `stdout`, a child's `print()` won't typically show up in the Parent's `stdout`.

`multiprocessing` is not the only way to create subprocesses. The `subprocess` module can be used to invoke commands, in which case the subprocess may also inherit `stdout`.

pip

The recommended tool for installing Python packages, and it comes automatically installed with Python.

It is used to install packages from PyPI (Python Package Index), or directly from source:

```
sudo pip install pep8

# Can install branches, tags, commit hashes.
sudo pip install git+https://github.com/PyCQA/flake8@main
```

pip can also install directly from distribution files:

```
# source distribution installation (sdist)
pip install sampleproject-1.0.tar.gz

# wheel distribution installation (wheel)
pip install sampleproject-1.0-py3-none-any.whl
```

Locally cloned repositories can be installed in editable mode, which is useful for development:

```
cd sampleproject
sudo pip install -e .
```

By default, all the pip operations apply globally. A separate environment with different packages and versions can be created using `virtualenv`. When a venv is active, all the pip operations will apply only on that venv:

```
ubuntu@ubuntu:~$ sudo pip install virtualenv
ubuntu@ubuntu:~$ virtualenv foo
ubuntu@ubuntu:~$ . ./foo/bin/activate
(foo) ubuntu@ubuntu:~$ pip install flake8
(foo) ubuntu@ubuntu:~$ deactivate
ubuntu@ubuntu:~$
```

Some projects that are using `pbr` (Python Build Reasonableness) contain a `requirements.txt` file, which contains a list of dependencies necessary for the project to run correctly. If `pbr` is not used, the dependencies are typically listed in the project's `setup.py` file. The dependencies can be unpinned or have restrictions:

```
oslo.cache
oslo.concurrency==5.0.1
oslo.config>=9.0.0
oslo.context>=5.0.0,<=6.0.0
oslo.db>=12.1.0,!=12.1.1
```

The dependencies can have their own dependencies and required versions. In this case, pip will find suitable versions so that all the dependencies have their requirements met. If there are conflicting dependency requirements, pip will return an error.

This list of requirements can then be installed by using the `-r` option:

```
pip install -r requirements.txt
```

OpenStack also maintains a list of upper constraints for requirements for all projects, which should be taken into account by the OpenStack projects (https://github.com/openstack/requirements/blob/master/upper-constraints.txt).

The upper constraints can be enforced by having the `UPPER_CONSTRAINTS_FILE` environment variable while installing the requirements: `pip install -r requirements.txt`.

`pip freeze` will list all of the currently installed libraries and their versions in `requirements` format. This can be useful when python applications are broken due to faulty dependencies:

```
ubuntu@ubuntu:~$ pip freeze
alembic==1.8.1
amqp==5.1.1
aniso8601==9.0.1
...
```

To uninstall a package, run:

```
pip uninstall sampleproject
```

Linting

Linters are static source code analysis tools that can be used to ensure minimum code quality requirements on projects. They can be used to make sure that the coding standards are met, prevents syntax errors, typos, bad formatting, and bad styling.

PEP8 (Python Enhancement Proposal) is a document containing guidelines and best practices on how to write Python code. PEP8 exists to improve the readability of Python code.

One of these tools is `flake8`, which is a wrapper that verifies `pycodestyle` (previously known as `pep8`), `PyFlakes`, and cyclomatic complexity.

It can be installed and used by running:

```
sudo pip install flake8
flake8 source_file_or_dir
```

An alternative linter is `pylint`. It follows the PEP8 recommended style, and it has a few additional features: checks that variable names are well-formed, checks that declared interfaces are truly implemented, and it can detect duplicate code. It can be installed and used by running:

```
sudo pip install pylint
pylint source_file_or_dir
```

`black` is another interesting tool that can be used. It's a Python code formatter that will update your code to apply the `pycodestyle` coding formatting standard:

```
sudo pip install black
black source_file_or_dir
```

Cookiecutter

Cookiecutter is a cross-platform CLI that creates projects from project templates (cookiecutters). It can be used for any type of project if the template is defined.

Projects like OpenStack has a few cookiecutter templates as well, depending on the type of project that needs to be created:

- https://opendev.org/openstack/cookiecutter.git
- https://opendev.org/openstack/ui-cookiecutter
- https://opendev.org/openstack/specs-cookiecutter - Smaller projects may have specs included in the project repository instead.
- https://opendev.org/openstack/oslo-cookiecutter

Cookiecutter can be installed through `pip`. The template file / URL can be passed to the `cookiecutter` CLI:

```
sudo pip install cookiecutter
cookiecutter https://opendev.org/openstack/cookiecutter.git
```

Cookiecutter will then go through a few project configuration options: module name, service name, repository name, project description, etc. After all the configuration options have been selected, a new folder is created, containing a git repository with the `Initial Cookiecutter Commit.` created. The project contains several key files:

Python project-specific files:

- `README.rst`, `CONTRIBUTING.rst`, `LICENSE`. The `README.rst` file should be updated with a more in-depth description. The `License` is Apache Version 2.0.
- `setup.cfg` and `setup.py`, containing the project metadata, which will be used when publishing to PyPI, or used by pip when installing the project: name, summary, author and its email, python requirements, packages.

- `requirements.txt`, containing a list of dependencies needed in order for the project to run correctly. OpenStack also maintains a list of upper constraints for requirements for all projects, which should be taken into account (https://github.com/openstack/requirements/blob/master/upper-constraints.txt). The upper constraints can be enforced by having the `UPPER_CONSTRAINTS_FILE` environment variable while installing the requirements: `pip install -r requirements.txt`.

- `test-requirements.txt`, containing a list of additional requirements needed for running the tests.

- `tox.ini` contains a list of tox environments (pep8, docs, releasenotes, etc.), and how to prepare them. Tox will typically create a virtualenv in which it will install the necessary requirements for that environment, and is the standard way to run the same types of jobs across multiple projects (e.g.: `tox -e pep8`)

- `.gitreview` contains details for OpenStack's Gerrit server. It contains the host, port, project, and branch name used whenever the command `git review` is used. Stable branches will have to update the branch name in this file.

- `releasenotes/` is a folder containing noteworthy release notes that will be published with each release of OpenStack. They typically include notes regarding new features, deprecation notices, upgrade impacts, etc.

- `doc/` is a folder containing various documentation for the project: installation, contributor guide, configuration, user / admin documentation, etc.

- `module_name/` is the folder that contains the actual source code of the project. It also contains the `tests/` folder and a base test class which can be extended. Most OpenStack projects define the `tests/unit/` and `tests/functional/` folders, and the file structure in the tests folders mirror the source code folder structure (e.g.: the subpackage `module_name/foo` will have its unit tests declared in `module_name/tests/unit/foo`).

There are various cookiecutter templates already defined for various types of projects, including non-Python ones. A few example templates can be seen here: https://www.cookiecutter.io/templates. Most of them contain the general boilerplate code required for a project, including things like: project metadata, dependency management tools, git pre-commit hooks, various linting and unit test jobs, Travis CI / GitHub actions jobs, publishing actions, documentation building and release notes jobs, etc.

Running tox

Tox aims to automate and standardize testing, packaging, and releasing Python software.

It is a `virtualenv` management and test command line tool that can be used for:

- checking that Python packages install and work correctly with different Python versions.
- running tests in each environment.
- frontend that can be used by CIs.

Tox is typically used by the OpenStack CIs to run various tests and checks on pull requests sent to projects. It can be installed by running:

```
sudo pip install tox
```

The OpenStack projects have a `tox.ini` file which is then used to run different types of jobs in those projects. The file contains a default `[testenv]` section, which applies to all tox environments (unless the tox environment overrides it), and different `[testenv:env-name]` sections. These sections contain a list of dependencies, environment variables to be set, and commands to be run. Example:

```
[testenv:docs]
deps = -r{toxinidir}/doc/requirements.txt
commands = sphinx-build -W -b html doc/source doc/build/html
```

Sample tox run:

```
tox -e pep8
tox -e py38
```

Generally, pep8 rules should not be ignored, but there are cases in which OpenStack projects will ignore some of them since the benefit they add can be questionable (e.g.: `W504 line break after binary operator`). These ignored rules can be added in the `tox.ini` file, in the `flake8` section:

```
[flake8]
ignore = E123,E125,W504
```

NOTE: Using pdb in code will cause tox to fail. In this case, the tests should be run manually:

```
# Activate the py38 environment, which has the necessary dependencies
installed.
. .tox/py38/bin/activate

# Run the test module which will reach the pdb breakpoints.
stestr run -n module_name.tests.test_file

# Deactivate the virtualenv.
deactivate
```

Unit testing

Unit testing is important in Python, being one of the most common and useful ways to catch early issues before getting a PR merged. Together with pep8 / flake8, a minimum code quality can be assured, as well as removing any potential coding mistakes. For example, it is expected of OpenStack developers to at least add enough test coverage for the newly introduced code.

Let's consider the following function:

```
sampler/utils.py:

def is_palindrome(string):
    return string == string[::-1]
```

We can then create the following unit test:

```
sampler/tests/unit/test_utils.py

import unittest

from foo import utils


class TestUtils(unittest.TestCase):

    def test_is_palindrome(self):
        string = "racecar"
        self.assertEqual(True, utils.is_palindrome(string))

        string = "foo"
        self.assertFalse(utils.is_palindrome(string))
```

Notes about unit tests:

- all unit tests **must** start with the `test` prefix, otherwise they are not run.
- `unittest.TestCase` has a few set up methods that are being called:
  - `setUpClass(cls)`:
  - `tearDownClass(cls)`:
  - `setUp`:
  - `tearDown`:

- `unittest.TestCase` has a few different assertions that can be used:
  - `assertEqual`
  - `assertTrue / assertFalse`
  - `assertDictEqual`
- other notable methods:
  - `addCleanup(self, callable)`:

Let's add the following files:

```
sampler/exceptions.py:

class SamplerException(Exception):
    pass


class RequestException(SamplerException):

    _msg = ("Request to %(url)s failed with status code %(status_code)s.
"
            "Reason: %(reason)s")

    def __init__(self, url, status_code=None, reason=None):
        super().__init__()
        self._url = url
        self._status_code = status_code
        self._reason = reason

    def __str__(self):
        return self._msg % {
            "url": self._url,
            "status_code": self._status_code,
            "reason": self._reason,
        }
```

```
sampler/requests.py

import json

import requests
from requests import exceptions as req_exc

from sampler import exceptions as exc

_DOG_URL = "http://dog-api.kinduff.com/api/facts"

def get_random_dog_fact():
    try:
        resp = requests.get(_DOG_URL)
        if resp.status_code != 200:
            raise exc.RequestException(_DOG_URL, resp.status_code,
resp.reason)

        return json.loads(resp.text)
    except req_exc.ConnectionError as e:
        raise exc.RequestException(_DOG_URL, reason=e)
```

Writing unit tests for this function can be trickier since we depend on a different library, which requires internet connection and an URL to download. In this scenario, we can mock any external dependencies:

```
sampler/tests/unit/test_utils.py

import json
from unittest import mock

from requests import exceptions as req_exc

from sampler import exceptions
from sampler import utils
from sampler.tests import base

class TestUtils(base.TestCase):

    @mock.patch("requests.get")
    def test_get_random_dog_fact(self, mock_get):
        fake_response = {
            "facts": ["foo"],
            "sucess": True,
        }
        mock_response = mock_get.return_value
        mock_response.text = json.dumps(fake_response)
        mock_response.status_code = 200
```

```
        fact = utils.get_random_dog_fact()

        self.assertEqual(fake_response, fact)
        mock_get.assert_called_once_with(utils._DOG_URL)

    @mock.patch("requests.get")
    def test_get_random_dog_fact_conn_error(self, mock_get):
        mock_get.side_effect = req_exc.ConnectionError

        self.assertRaises(exceptions.RequestException,
                          utils.get_random_dog_fact)
        mock_get.assert_called_once_with(utils._DOG_URL)

    @mock.patch("requests.get")
    def test_get_random_dog_fact_non_200(self, mock_get):
        mock_response = mock_get.return_value
        mock_response.status_code = 404

        self.assertRaises(exceptions.RequestException,
                          utils.get_random_dog_fact)
        mock_get.assert_called_once_with(utils._DOG_URL)
```

We can see that we're mocking `requests.get` for every single test. We can instead do it only once in the `setUp`:

```
sampler/tests/unit/test_utils.py

class TestUtils(base.TestCase):

    def setUp(self):
        super().setUp()

        patcher = mock.patch("requests.get")
        self._mock_get = patcher.start()
        self._mock_response = self._mock_get.return_value
        self.addCleanup(patcher.stop)
```

NOTE: In order for the new tests to be discoverable, the `__init__.py` file should be added to each child folder under `sampler/tests/`.

Unit tests should cover as much of the code as possible, and be kept as simple as possible. For this, we are using `unittest.mock` to mock any external calls. One noteworthy argument is the `autospec=True` argument, which should be used in order to prevent false positive function calls. For example, if we would have the following code:

```
def get_random_dog_fact():
    resp = requests.get(_DOG_URL, some_inexistent_arg="foo")
    return json.loads(resp.text)
```

The call `resp = requests.get(_DOG_URL, some_inexistent_arg="foo")` is obviously incorrect. But because `requests.get` is mocked in the tests, then that error wouldn't be caught, unless we use `autospec=True`. Adding `autospec=True` to every single mock can be quite verbose, and it can be forgotten. Instead, the mock module can be patched to autospec by default by adding the following code to `sampler/tests/base.py`, which should be used as a base for all unit tests:

```
from oslotest import mock_fixture

# NOTE(claudiub): this needs to be called before any mock.patch calls
are
# being done, and especially before any other test classes load. This
fixes
# the mock.patch autospec issue:
# https://github.com/testing-cabal/mock/issues/396
mock_fixture.patch_mock_module()
```

Exercises

Exercise 1 - Array slicing

R2-D2 wants to get back home, yet he encountered some problems. The text that will get him back home safely is in th
e wrong order. Who will be able to help R2-D2 get back home safely?

Task: Help R2-D2 get home by using array-slicing so that the final text will look like this: "R2-D2 va ajunge acasa"

```
text = ["acasa", "ajunge", "va", "nu", "R2-D2"]
```

Exercise 2 - If, range, for

Anakin and Obi-Wan are fighting over the number of vowels and consonants of the expression 'hodoronc-tronc'. Is
there a chance of this fight to get to an end?

Task: Your task is to write a python script that will calculate the number of vowels and consonants of the expre
ssion 'hodoronc-troncusingif, range, for` learnt previously.

```
text = "hodoronc-tronc"
vowels = "aeiou"
```

Exercise 3 - List comprehension

Jabba just started learning python and he discovered something called `list comprehension`. He wishes to create a list, which will contain all the even numbers from `[1, 2, 3, 4, 5, 6, 2, 8, 6, 5, 3, 1, 9]` using this new technique that he just learnt. How should he do this?

Task: Write a python script that generates a list containing all the even numbers only once from Jabba's list using list comprehension.

Hint: Jabba's best friend is called Set.

```
Jabba_sir = [1, 2, 3, 4, 5, 6, 2, 8, 6, 5, 3, 1, 9]
```

Exercise 4 - Dictionaries, files and imports

Leia just received a very hard task, which she must solve in 5 minutes. She must parse a JSON file and extract some information. Can you help her do this?

Task: You get a JSON file `data.json` containing different data about employees from a company. You have to parse the JSON and print the names of the employees as well as the department they work in like this: `Name: John Doe | Department: MMORPG Farming`.

```
{
  "employees": [
    {
      "id": 1,
      "name": "Han Solo",
      "department": "Engineering"
    },
    {
      "id": 2,
      "name": "Padme Amidala",
      "department": "Marketing"
    },
    {
      "id": 3,
      "name": "Bobba Fett",
      "department": "Finance"
    }
  ]
}
```

Exercise 5 - functions:

Task: Create a function `copy_fields` that returns a new dictionary that will only have the wanted fields from a given dictionary. The function will have the following parameters: the dictiornary to be copied from, an optional default value if the field cannot be found in the dictionary, and a variable list of arguments representing the fields to copy.

Exercise 6 - decorators:

Task: Create an Exception catcher decorator, which will catch all the exceptions, log them, and return `None` instead of having the exception be uncaught.