Advanced Python

1 Creating New Classes

A class is a blueprint to create new objects, and objects are responsible for storing data and functions together. Creating classes is an useful abstraction, as it allows us to use common patterns to simplify code. For example, in game theory a useful abstraction could be the player of the game. We can use a class to create such a player, add a state representing his utility, a method for computing his utility, a method for his possible actions, and so on.

To create a class we use the following syntax:

```
class ClassName:
    # variables shared across all class instances
    var1 = 'abc'
    var2 = 123

    # Constructor: function that is exectued
    # when the class is instantiated
    def __init__(self, input1):
        # self represents the instance of the object

        # create a variable for the instance, and store the value
        # of args[0]
        self.a = input1

    # Method of the class
    def printVariable(self):
        # a method has access to all the variables of the
        # class instance (the object) via the keyword self
        print(self.a)
    # The methods of a class almost always have self as the first input
    # There are cases where the input will be cls, and there are cases
    # where there will be no input. We will talk about this after the
    # Decorator section.

    def incrementVariable(self):
        # a method can modify existing variables and create new ontes
        self.a = self.a + 1
        print(self.a)
        self.b = 10*self.a

    def computeSum(self):
        return self.a + self.b
```

```
    def createVariable(self, c):
        self.c = c
        return self.a + self.b + self.c

    def printClassVariable(self):
        print(self.var1, self.var2)
```

We can now create an object instance from the blueprint `ClassName`. That is, we can instantiate `ClassName`:

```
instance = ClassName(3.1415)
# Use its methods
instance.printVariable()
instance.incrementVariable()
print(instance.createVariable(14))
instance.printClassVariable()
# We can directly access the variables
print(instance.a, instance.b, instance.c)
print(instance.var1, instance.var2)
```

All classes are based on the blueprint of an even more general abstract class. They inherit some attributes and methods from this abstract class. One attribute is named `__dict__`, which stores all of the variables in the instance.

```
print(instance.__dict__)
# It also has a method named __setattr__
print(instance.__setattr__)
# This method is what is used when we assign the value of a variable
# using the symbol: =
# All it does is update the variables in
# the __dict__ with the new value.

# We can see the other attributes and methods with dir
dir(instance)
```

2 Creating a Class for Stock Prices

Let's create a new class that will represent the stock of a company:

```
import numpy as np
import matplotlib.pyplot as plt


# Create a new class to represent a Stock
class Stock:
    frequencies = ['1min', '5min']

    def __init__(self, p0, sigma, mean=0.0, frequency='5min'):
        """
        """
        # Validate inputs
        if p0 < 0:
```

```
            raise ValueError("Negative initial price.")
        if sigma <= 0:
            raise ValueError("Non-positive price standard deviation.")
        if frequency not in frequencies:
            raise ValueError(
                f"Invalid frequency. Choose: {', '.join(frequencies)}")
        # Initialize class instance variables:
        # current log-price
        self.p = np.log(p0)
        # standard deviation of log-prices:
        self.sigma = sigma
        # average log-return
        self.mu = mean
        # set frequency of price updates
        self.frequency = frequency
        if frequency == '5min':
            # there are 79 5-minutes intervals from 9:30 AM to 4 PM
            self.delta = 1/79
        elif frequency == '1min':
            self.delta = 1/391
        # history of log-prices
        self.prices = [self.p]

    def timeStep(self):
        """Updates the stock price as if a delta time unit had passed.
        Log-prices are simulated from a Brownian motion:
        p_t = p_{t-1} + mu*delta + sigma*sqrt(delta)*Normal
        """
        trend = self.mu*self.delta
        ret = trend + self.sigma*np.sqrt(self.delta)*np.random.normal()
        self.p += ret
        self.prices.append(self.p)

    def timeSteps(self, total_steps):
        """Updates the stock price as if a number of time steps of size
        delta had passed.
        """
        for i in range(total_steps):
            self.timeStep()

    def getPrice(self):
        return np.exp(self.p)

    def printPrice(self):
        print(f'Current stock price: US$ {np.exp(self.p)}')

    def graphHistory(self):
        fig, ax = plt.subplots()
        ax.plot(range(len(self.prices)), np.exp(self.prices),
                color='black', linewidth=0.5)
        ax.set(title='Evolution of Stock Price',
```

```
                xlabel=f'Time (in {self.frequency} intervals)',
                ylabel=r'Price (in US$)')
        current_price = ax.scatter(len(self.prices), np.exp(self.p),
                                   marker='o', s=100)
        if self.p > self.prices[0]:
            current_price.set_color('green')
        else:
            current_price.set_color('red')
        ax.grid(True)
        ax.set_xlim(left=0)
        fig.show()
```

```
spy = Stock(296.46, 0.011)
spy.printPrice()
spy.timeStep()
spy.printPrice()
spy.timeSteps(79*252)
spy.printPrice()
spy.graphHistory()
```

3 Underscores and Dunders in Names

In a some of the examples, you saw the following attributes:

```
spy.__dict__
spy.__class__
spy.__doc__
```

The attributes use underscores in their names. It turns out the use of underscores in the names of variables has meaning. Underscores can be used in 5 different ways:

1. Single Underscore Leading the Variable Name: `_var`
2. Single Underscore Trailing the Variable Name: `var_`
3. Double Underscore Leading the Variable Name: `__var`
4. Double Underscore Leading and Trailing the Variable Name: `__var__`
5. Single Underscore by Itself: `_`

Let's explore each of the use cases.

3.1 Single Underscore Leading the Variable Name: `_var`

The `_var` type of naming is a convention (specified in PEP8) used by Python programmers to indicate that the variable (or method) is meant for internal use only. That is, whatever is associated with `_var` (an attribute or a method) should only be used by functions inside the class that defined it, and is not meant to be accessed directly outside the class definition. It is only a convention, so it is not enforced by the interpreter. However, if you see a variable named this way, you should avoid messing with it since it might be used internally in the class for other things.

```
class SingleUnderscoreLeading:
    def __init__(self):
        self.name = "ABC"
        self._private = "DEF"

    def _internal_function(self):
        self._private += "GHI"


x = SingleUnderscoreLeading()
print(x.name)
print(x._private)
print(x._internal_function())
```

3.2 Single Underscore Trailing the Variable Name: `var_`

The use of a trailing underscore in the name of a variable is when you need to give a name to a variable, but that name is already taken by a keyword in Python.

```
class Asset:
    def __init__(self, type_):
        # type is a keyword, so use type_ instead
        self.type_ = type_
        # type of the asset: Stock, Option, Debenture, ...
```

This is also a convention and is specified in PEP8.

3.3 Double Underscore Leading the Variable Name: `__var`

The use of a double underscore at the beginning of a variable makes the Python interpreter to change the name of the variable (attribute or method) when the class is instantiated. The change of names is called name mangling, and it serves two purposes:

- Making the variable harder to access outside of the class definition. This is similar to the case of `_var`, but now it is actually enforced by the Python interpreter.
- Making it easier to avoid name conflicts if you use this class as a base for another class (subclass). If your class and subclass have many variables, it can be easy to have naming conflicts, so name mangling helps avoid that.

Example:

```
class NameMangling:
    def __init__(self):
        self.a = 1
        self._b = 2
        self.__c = 3

test = NameMangling()
print(test.a)
print(test._b)
print(test.__c)                  # no attribute named __c
# of course, the __c name was mangled
```

```
dir(test)
# __c was substituted by _NameMangling__c
# It uses a single leading _, indicating it is reserved
# for internal use only.
# But, we can access it outside of the class if we
# really want (although we shouldn't).
print(test._NameMangling__c)
```

The object's methods can access the variables without any issues:

```
class ManglingAMethod:
    def __init__(self):
        self.a = 1
        self._step_size = 2

    def __increase(self):
        self.a += self._step_size

    def increase(self, total_steps=1):
        for i in range(total_steps):
            self.__increase()


test = ManglingAMethod()
print(test.a)
test.__increase()               # no attribute __increase
print(test.a)
test.increase()
print(test.a)
test.increase(5)
print(test.a)
dir(test)
# although the name was mangled after the class is instantiated
# the methods of the class can access it normally
```

3.4 Double Underscore Leading and Trailing the Variable Name: `__var__`

Variable names with double underscores are known as dunders, where dunder stands for double underscore. For example the `__init__` method of a class is a dunder method.

Names with double underscores do not go through name mangling. However, they have a special meaning in the Python language. These dunder variables define special operations in Python. For example, `__init__` defines the class constructor, `__doc__` stores the class documentation, `__setattr__` defines what the = operator should do, and so on. There are several of these dunder attributes and they are all discussed in the Python data model page. We will discuss a few of them in the next section.

You should avoid defining new attributes of a class using double underscores, since they can be used by Python in a special way. Even if they are not being used right now, in the future something might be changed in the language specification and the name you used could start having a special meaning.

3.5 Single Underscore by Itself: _

A single underscore is actually used outside of the context of classes and objects. It is used to represent some output of a function that we do not care about. Say you have a function that returns two outputs, but you do not care about the second output. Then, when you unpacked the function results, you would use _ to store the value you do not care about. In other words, _ represents throwaway values. Example:

```
def foo(x):
    return x**2, np.sqrt(x)


# We only want the 1st output
a = foo(4)
# but a function will always return all of its outputs in a tuple
# We can use tuple unpacking
a, b = foo(4)
# Say we do not really care about b, then substitute it for _
a, _ = foo(4)
# Or, if we do not care about the value of a, then we substitute it for _
_, b = foo(4)


person = ('First Name', 'Last Name', 29, 0.0)
_, last_name, age, _ = person
print(last_name, age)
print(_)
```

The _ indicates values that we do not have an use for, and serves as a temporary variable. In some REPLs, _ also represents the result of the last expression evaluated and that was not assigned a name.

4 Special Methods

We will now explore some of the dunder methods of a class in more detail.

4.1 The __len__ method

When we create a list in Python, we can use the function `len` to get its length. Remember that everything in Python is an object, including lists. When we call `len` on an object, the object instance will look for the method `__len__` and execute, and then return its value. If the dunder method `__len__` is not implemented, it will lead to an exception.

```
a = [1, 2, 3]
print(len(a))
dir(a)
print(a.__len__())


class MyList:
    def __init__(self, *values):
        self.values = list(values)
```

```
a = MyList(1, 2, 3, 4, 5)
a.values
# Try to find the length of the object a:
len(a)
# error, because there is no __length__ method implemented


class MyList:
    def __init__(self, *values):
        self.values = list(values)

    def __len__(self):
        # len is implemented for the list type, which is self.values
        return len(self.values)


a = MyList(1, 2, 3, 4, 5)
print(len(a))
```

4.2 The __bool__ method

The __bool__ method is responsible for returning a boolean when we evaluate an object in the if context, or when we call the built-in bool on the object:

```
bool([])
bool([1, 2, 3])


class Answer:
    def __init__(self):
        self.answer = input("What do monkeys eat? ")

    def __bool__(self):
        if self.answer in set(["Bananas", "bananas", "BANANAS!!!!"]):
            return True
        else:
            return False


my_answer = Answer("bananas")
print(bool(my_answer))
my_answer = Answer("apples")
print(bool(my_answer))
```

4.3 The __call__ method

The __call__ method is what makes an object callable. That is, the object instance can be behave as a function.

```
class PutOption:
    def __init__(self, stock_price, sigma, strike, tenor):
        # Validate inputs
        if stock_price <= 0:
            raise ValueError("Stock price must be positive.")
        if sigma <= 0:
            raise ValueError("Standard deviation must be positive.")
        if strike <= 0:
            raise ValueError("Strike price must be positive.")
        if tenor <= 0:
            raise ValueError(
                "Time to expiration (in days) must be positive.")
        # Underlying asset
        self.underlying = Stock(stock_price, sigma, frequency='5min')
        # Strike price
        self.strike = np.float(strike)
        # Time to expiration in days
        self.tenor = tenor

    def __payoff(self, price):
        return np.max((0, self.strike - price))

    def getPayoff(self):
        # Simulate stock price
        self.underlying.timeSteps(self.tenor*79)
        # Return option payoff
        return self.__payoff(self.underlying.getPrice())

    def __call__(self):
        """Simulates stock price and returns payoff."""
        return self.getPayoff()


put = PutOption(200, 0.011, 220, 30)
put()
```

4.4 The __repr__ and __str__ methods

The `__repr__` method is used when we inspect an object on the REPL, or when we call the built-in `repr` on an object. The `__str__` method is used when we call `print` on the object. The `print` function calls `str` on the object, which gives us the result of the object's `__str__` method. The differences are discussed below:

```
x = [1, 2, 3]
x                                   # or repr(x)
# what about print?
print(x)                            # calls str(x)
y = {'a': 1, 'b': 2}
repr(y)
print(y)
```

`repr` is used to give an unambiguous representation of the object. In the case of simple objects, `repr` and `str` coincide. `str` is used to give a human readable representation of the object

Consider a more complicated object:

```
import datetime
# Get a date object representing today's date
today = datetime.date.today()
# print will give a readable representation of the object:
str(today)
# but repr will give an unambiguous representation of the object
repr(today)
# notice that repr basically returns the way we construct the object
# indeed:
today = datetime.date(2019, 7, 9)
print(today)
```

We can define the `__repr__` and `__str__` methods ourselves:

```
class PutOption:
    def __init__(self, stock_price, sigma, strike, tenor):
        # Validate inputs
        if stock_price <= 0:
            raise ValueError("Stock price must be positive.")
        if sigma <= 0:
            raise ValueError("Standard deviation must be positive.")
        if strike <= 0:
            raise ValueError("Strike price must be positive.")
        if tenor <= 0:
            raise ValueError(
                "Time to expiration (in days) must be positive.")
        # Underlying asset
        self.stock_price = stock_price
        self.sigma = sigma
        self.frequency = '5min'
        self.underlying = Stock(stock_price,
                                sigma,
                                frequency=self.frequency)
        # Strike price
        self.strike = np.float(strike)
        # Time to expiration in days
        self.tenor = tenor

    def __payoff(self, price):
        return np.max((0, self.strike - price))

    def getPayoff(self):
        # Simulate stock price
        self.underlying.timeSteps(self.tenor*79)
        # Return option payoff
        return self.__payoff(self.underlying.getPrice())

    def __call__(self):
```

```
        """Simulates stock price and returns payoff."""
        return self.getPayoff()

    def __repr__(self):
        return (f"PutOption({self.stock_price}, {self.sigma}"
                f", {self.strike}, {self.tenor})")

    def __str__(self):
        return (f"PutOption: Strike=US${self.stock_price}, "
                f"Tenor={self.tenor} days")


put = PutOption(200, 0.011, 220, 30)
repr(put)
str(put)
```

5 Decorators

Decorators are used to modify callables (like functions, methods or classes), but without permanently changing them. That is, decorators perform local changes to a callable.

5.1 Input Validation

In the classes above we did some data validation. It is common to have several functions where you need to validate inputs (or perform assertions) that are very similar. For example:

```
def ols(x, y):
    if x.shape[0] != y.shape[0]:
        raise ValueError("Number of rows must match for x and y")
    if x.shape[0] < x.shape[1]:
        raise ValueError(
            "x: Need more observations than charactersitics.")
    # estimate ols
    # return estimates
    return


def probit(x, y):
    if x.shape[0] != y.shape[0]:
        raise ValueError("Number of rows must match for x and y")
    if x.shape[0] < x.shape[1]:
        raise ValueError(
            "x: Need more observations than charactersitics.")
    # estimate probit
    # return estimates
    return


def logit(x, y):
```

```
    if x.shape[0] != y.shape[0]:
        raise ValueError("Number of rows must match for x and y")
    if x.shape[0] < x.shape[1]:
        raise ValueError(
            "x: Need more observations than charactersitics.")
    # estimate logit
    # return estimates
    return


x = np.random.random((5, 2))
y = np.random.random((5, 1))
ols(x, y)
x = np.random.random((4, 2))
ols(x, y)
x = np.random.random((2, 5))
ols(x, y)
```

We repeated ourselves in the code above. There is a common pattern of validation that we should be able to simplify. For example, we could have one function that validates the input:

```
def validateRegressionInput(x, y):
    if x.shape[0] != y.shape[0]:
        raise ValueError("Number of rows must match for x and y")
    if x.shape[0] < x.shape[1]:
        raise ValueError(
            "x: Need more observations than charactersitics.")
    return True


def ols(x, y):
    if validateRegressionInput(x, y):
        # estimate ols
        # return estimates
    return


def probit(x, y):
    if validateRegressionInput(x, y):
        # estimate probit
        # return estimates
    return


def logit(x, y):
    if validateRegressionInput(x, y):
        # estimate logit
        # return estimates
    return
```

Now our functions are much cleaner and do input validation without repeating code. An alternative solution is to wrap the functions in another function which performs the

validation. That is, create a function that returns a function, and this new function will do input validation before estimating the regression:

```
def validateRegressionInput(func):
    def functionWithValidatedInput(x, y):
        if x.shape[0] != y.shape[0]:
            raise ValueError("Number of rows must match for x and y")
        if x.shape[0] < x.shape[1]:
            raise ValueError(
                "x: Need more observations than charactersitics.")
        return func(x, y)
    return functionWithValidatedInput
```

The function `validateRegressionInput` is a functional in the math sense. That is, we pass it a function, and it returns a new function. Observe that this new function takes the same inputs as the original function: x and y. However, the function returned by `validateRegressionInput` also validates the inputs. After validating the inputs, the original function is called (here represented by `func`). `func` is called with the inputs x and y, and it returns the value of `func(x, y)`.

`validateRegressionInput` takes a function and returns a similar function that has been extended to do input validation:

```
# We can now define our functions
def ols(x, y):
    # estimate ols
    # return estimates
    return


# Now we call the functional validateRegressionInput to get back an extende
# of ols, where the input is validated!
ols = validateRegressionInput(ols)


def probit(x, y):
    # estimate probit
    # return estimates
    return


probit = validateRegressionInput(probit)


def logit(x, y):
    # estimate logit
    # return estimates
    return


logit = validateRegressionInput(logit)


x = np.random.random((5, 7))
```

```
y = np.random.random(5)
logit(x, y)
```

Here we see a common pattern: create a function and then modify it. Now, Python has a syntax for this pattern, and this syntax is called the decorators syntax. A decorator is a function that returns another function, just like `validateRegressionInput` in the example above. We can then decorate other functions with this decorator, that is, extend other functions with the decorator.

```
# THIS:
def ols(x, y):
    # estimate ols
    # return estimates
    return


ols = validateRegressionInput(ols)

# IS EQUIVALENT TO:
@validateRegressionInput
def ols(x, y):
    # estimate ols
    # return estimates
    return


x = np.random.random((5, 2))
y = np.random.random(5)
ols(x, y)
y = np.random.random(6)
ols(x, y)
```

5.2 Timing Execution

Consider a scenario where you have four different functions to compute some task, and you would like to know which is faster.

```
import random
import math
import numpy as np


def f1(sample_size):
    sample = []
    for i in range(sample_size):
        sample.append(random.gauss(0, 1))
    return sum(sample)/len(sample)


def f2(sample_size):
    sample = []
```

```
    for i in range(sample_size):
        sample.append(random.gauss(0, 1))
    return math.fsum(sample)/len(sample)

def f3(sample_size):
    sample = [random.gauss(0, 1) for _ in range(sample_size)]
    return math.fsum(sample)/len(sample)


def f4(sample_size):
    sample = np.random.normal(size=(sample_size))
    return sample.mean()
```

Notice the use of the built-in `random` package for generating random numbers, and the use of the built-in `math` package and its function `fsum` to accurately compute the sum of the list of numbers (more accurate than `sum` in terms of numerical errors).

We can time the execution of functions with the functions from the built-in `time` module.

```
import time
# Get a value measured in seconds of a "performance counter"
# (a clock with a good resolution for measuring time)
start = time.perf_counter()
# The number itself does not mean anything, we need to compare
# it to the stop.
# time to get the elapsed time in seconds
stop = time.perf_counter()
print(f"Elapsed time: {stop - start: .4f} seconds")
```

Let's use `time.perf_counter` to build a decorator for measuring the execution time of a function:

```
def time_this(func):
    def wrapper(sample_size):
        start = time.perf_counter()
        results = func(sample_size)
        stop = time.perf_counter()
        print(f"Elapsed time: {stop - start: .4f} seconds")
        return results
    return wrapper
```

We can decorate the functions we want to time, and when we call them, the time will be printed automatically.

```
@time_this
def f1(sample_size):
    sample = []
    for i in range(sample_size):
        sample.append(random.gauss(0, 1))
    return sum(sample)/len(sample)


@time_this
def f2(sample_size):
```

```
    sample = []
    for i in range(sample_size):
        sample.append(random.gauss(0, 1))
    return math.fsum(sample)/len(sample)


@time_this
def f3(sample_size):
    sample = [random.gauss(0, 1) for _ in range(sample_size)]
    return math.fsum(sample)/len(sample)


@time_this
def f4(sample_size):
    sample = np.random.normal(size=(sample_size))
    return sample.mean()


for f in [f1, f2, f3, f4]:
    f(10000)
```

We can modify the decorator so that it tells us which function is being executed:

```
def time_this(func):
    def wrapper(sample_size):
        start = time.perf_counter()
        results = func(sample_size)
        stop = time.perf_counter()
        print(f"Function: {func.__name__}")
        print(f"Elapsed time: {stop - start: .4f} seconds")
        return results
    return wrapper


@time_this
def f1(sample_size):
    sample = []
    for i in range(sample_size):
        sample.append(random.gauss(0, 1))
    return sum(sample)/len(sample)


@time_this
def f2(sample_size):
    sample = []
    for i in range(sample_size):
        sample.append(random.gauss(0, 1))
    return math.fsum(sample)/len(sample)

@time_this
def f3(sample_size):
    sample = [random.gauss(0, 1) for _ in range(sample_size)]
```

```
    return math.fsum(sample)/len(sample)


@time_this
def f4(sample_size):
    sample = np.random.normal(size=(sample_size))
    return sample.mean()


for f in [f1, f2, f3, f4]:
    f(10000)
```

5.3 Multiple Decorators

We can use multiple decorators to decorate a function. Since each decorator returns a function, we can chain more than one, and, at the end, we will still have a function.

```
def capitalize_all(func):
    def wrapper(name):
        return func(name.upper())
    return wrapper


def yell(func):
    def wrapper(name):
        greeting = func(name)
        return f"{greeting}!!!!!!!!!!!!!!!!!!!!!!"
    return wrapper


@capitalize_all
def greet(name):
    return f"Hello, {name:s}!"


greet("Everyone")


@yell
@capitalize_all
def greet(name):
    return f"Hello, {name:s}!"


greet("Everyone")
```

5.4 Decorating Functions that Have Many Arguments

We can use the `*args` and `**kwargs` notations to deal with functions of many arguments:

```
# Create a decorator for a function that can take many input arugments
```

```
def decorator(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper
```

Remember that `*args` collects all positional inputs and stores them in the tuple `args`, which is available inside the `wrapper` function. And `**kwargs` collects all keyword arguments and stores them in the dictionary `kwards`, which is also available inside the `wrapper` function.

The `args` tuple and the `kwargs` dictionary are then forwarded to the original function `func`. The `*` and `**` are unpacking operators in this context (the symbol `*` and `**` are overloaded!). When we call `func(*args, **kwargs)`, the operator `*` will unpack `args`, so that each input in the tuple `args` is passed to `func` separately. The operator `**` will unpack `kwards`, so that each keyword input is also passed separately. `func` can then function regularly with its inputs.

```
def decorator(func):
    def wrapper(*args, **kwargs):
        print('Positional arguments: ', args)
        print('Keyword arguments: ', kwargs)
        return func(*args, **kwargs)
    return wrapper


@decorator
def foo(x, y, z=1):
    print(x, y, z)


foo(1, 2, z=10)
foo(1, 2, 3)
foo(z=1, x=2, y=-3)
```

5.5 Making the Documentation of Decorated Functions Available

When we create a function with a docstring, we can call `help` to get back useful comments about the function.

```
def greet(name):
    """Returns a warm welcome message."""
    return f"Hello, {name}!"


# Function name and documentation are available
greet("abc")
greet.__name__
greet.__doc__
```

However, when we decorate a function, we loose the original docstring:

```
# Decorate greet
def toUpperCase(func):
```

```
    def wrapper(name):
        """Decorator docstring"""
        return greet(name).upper()
    return wrapper


decorated_greet = toUpperCase(greet)
print(decorated_greet("Abc"))
# However, the original function name and documentation are gone
decorated_greet.__name__
print(decorated_greet.__doc__)
```

We can decorate the original function `func` with the decorator `wraps` from the Python module `functools`. By decorating `func` with `wraps` before calling `wrapper`, we can make the original docstring available in the decorated version:

```
import functools


def toUpperCase(func):
    @functools.wraps(func)
    def wrapper(name):
        """Decorator docstring"""
        return greet(name).upper()
    return wrapper


decorated_greet = toUpperCase(greet)
decorated_greet.__name__
decorated_greet.__doc__
```

6 Instance, Class and Static Methods

There are two special decorators that can be used inside a class definition: `classmethod` and `staticmethod`. These decorators are built-in the language and modify the methods of a class.

```
class DecoratedClass:
    def instance_method(self):
        # has access to the variables of the instance
        # and also the variables common to all instances (class variable)
        # via self.__class__
        return "instance method", self

    @classmethod
    def class_method(cls):
        # instead of receiving self, it receives cls
        # which represents the class itself
        # this method does not have access to the instance variables
        # only the class variables
        # That is, cls points to the class itself.
```

```
        return "class method", cls

    @staticmethod
    def static_method():
        # this method does not have self neither cls as input
        # it cannot access the instance variables nor the class variables
        # but we can define it to take other inputs
        return "static method"
```

The `instance_method` can access the object's instance variables via `self`. The `class_method` can access the class variables via `cls`, that is, it can look at the blueprint of the object itself. The `static_method` cannot access any of the instance or the class variables.

```
obj = DecoratedClass()
print(obj.instance_method(), obj)
print(obj.class_method(), DecoratedClass)
print(obj.static_method())
```

Let's try to use the methods of `DecoratedClass` without an instance:

```
DecoratedClass.static_method()
# works, since it does not depend on anything else, we just need
# the class to be defined
DecoratedClass.class_method()
# also works, because it only depends on the class itself, and not on a
# particular instance
DecoratedClass.instance_method()
# does not work, because it needs an instance to access
```

The `staticmethod` decorator basically makes the method a regular function. It is used to make that function live in the namespace of the class.

The `classmethod` decorator gives a method access to the class itself. This can be used to create extra constructors!

```
class Option:
    def __init__(self, strike, tenor, type_):
        self.strike = strike
        self.tenor = tenor
        self.type_ = type_

    def __repr__(self):
        return f"Option({self.strike}, {self.tenor}, {self.type_})"

    @classmethod
    def long_straddle(cls, strike, tenor):
        return cls(strike, tenor, 'long-straddle')

    @classmethod
    def short_straddle(cls, strike, tenor):
        return cls(strike, tenor, 'short-straddle')

    def payoff(self, price):
        if self.type_ == 'c':
```

```
            return np.max([0, price - self.strike])
        elif self.type_ == 'p':
            return np.max([0, self.strike - price])
        elif self.type_ == 'long-straddle':
            return np.abs(price - self.strike)
        elif selt.type_ == 'short-straddle':
            return -np.abs(price - self.strike)


call = Option(200, 30, 'c')
repr(call)
long_straddle = Option.long_straddle(200, 30)
repr(long_straddle)
```

7 Iterators

Iterators are objects that implement the `__iter__` and `__next__` dunder methods. They represent a collection of things we want to iterate over. We have used iterators in for-loops:

```
for i in [0, 1, 2]:
    print(i)
# the list [0, 1, 2] is an iterator
# it is a collection of things we want to iterate over
# and it implements the dunder methods iter and next
```

Another example are file objects, which could be used to iterate over the lines of a file.

```
with open('my_file.txt', 'w') as f:
    f.write('\n'.join(["a","b","c"]))
with open('my_file.txt', 'r') as f:
    for line in f:
        print(line)
```

An iterator represent a collection that we can iterate over (with a for-loop, for example). To make it work, the iterator object must have a `__next__` method, that returns the next value in the set:

```
f = open('my_file.txt', 'r')
print(f.__next__())
# Each time we call __next__ the object returns the next value
# in the collection.
# Same as:
print(next(f))
# we can call the dunder next until the last line
print(f.__next__())
# if we call it again, we will get an exception
print(f.__next__())
```

Notice that we get an exception with an interesting name: `StopIteration`. This means the iterator is over, there are no more elements in the collection to iterate over.

Let's see how a for-loop really works in Python:

```
# We can go back to the beginning of the file by calling
f.seek(0)

# Consider this loop:
for line in f:
    print(line)

f.seek(0)
# Under the hood, what is happening is:
while True:
    try:
        line = next(f)
    except StopIteration:
        break
    else:
        print(line)
```

That is, read a line, then run the code that was inside the for-loop. Then, read the next line and so on. When we get the StopIteration exception, it means we are out of lines to read, so we break the loop.

Another example of iterator is the object returned by enumerate:

```
names = ["A", "B", "C"]
numbered_names = enumerate(names)
print(next(numbered_names))

# for-loop with enumerate:
for pos, name in enumerate(names):
    print(f"{pos}: {name}")


# Equivalent to:
iterator = enumerate(names)
while True:
    try:
        pos, name = next(iterator)
    except StopIteration:
        break
    else:
        print(f"{pos}: {name}")
```

Notice that after getting all elements of the iterator, we reach the end and `StopIteration` is raised. We cannot reset `numbered_names` like we could with the file. The file object is a special case.

To iterate over `numbered_names` from the beginning, we would need to call enumerate again. That is, calling enumerate is, somehow, giving us an iterator.

Now, an **ITERABLE** (as opposed to **ITERATOR**), is an object that gives us an iterator when we ask for it. Again, an iterable is an object that gives us an iterator.

We can ask for an iterator from an iterable by using the function iter, or the dunder method `__iter__`. For example, a list itself is not an iterable, but it is an iterator.

```
x = [0, 1, 2, 3]
next(x)
```

```
# error, list is not an iterator
```

But why can we use lists in a for-loop? Because it is an iterable, that is, it can create an iterator from the values in the list:

```
x_iterator = iter(x)
next(x_iterator)
# works!
```

A list is an object that can generate iterators. It is an iterable. Thus, when we use a for-loop with an ITERABLE, what is happening is:

```
iterator = iter(iterable)
while True:
    try:
        value = next(iterator)
    except StopIteration:
        break
    else:
        # do something with value
        pass
```

We use the iterable to get an iterator. And then loop over the elements of the iterator until it is done. The iterator is then **SPENT**, that is, there are no more values to get out of it. If we want to loop over again, then we need to create a fresh iterator, by asking the iterable for a new iterator. The list is an iterable that saves the elements, and can generate as many iterators as we need from the same values.

Let's create a simple iterable. An iterable must implement the `__iter__` method to return an iterator.

```
class FibonacciIterable:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        return FibonacciIterator(self.n)


# Now, we need to creat the ITERATOR, which implements
# the __next__ method.
class FibonacciIterator:
    def __init__(self, n):
        self.n = n
        self.previous = [1]

    def __next__(self):
        value = self.previous[-1]
        if value > self.n:
            raise StopIteration
        else:
            next_value = sum(self.previous[-2:])
            self.previous.append(next_value)
            return value
```

```
iterable = FibonacciIterable(10)
iterator = iter(iterable)
next(iterator)
```

We can use an iterable in a for-loop:

```
for fib in FibonacciIterable(10):
    print(fib)

# equivalent to:
iterator = FibonacciIterable(10).__iter__()
while True:
    try:
        fib = next(iterator)
    except StopIteration:
        break
    else:
        print(fib)
```

Notice that we are repeating oruselves with the code above. We had to create two separate objects, an iterable and an iterator. We can create a single object that implements both the `__iter__` method and the `__next__` method:

```
class FibonacciSequence:
    def __init__(self, stop_at_most):
        self.stop_at_most = stop_at_most
        self.previous = [1]

    def __iter__(self):
        # the instance itself implements the __next__ method, so itself
        # is an iterator (and an iterable)
        return self

    def __next__(self):
        value = self.previous[-1]
        if value > self.stop_at_most:
            raise StopIteration
        else:
            next_value = sum(self.previous[-2:])
            self.previous.append(next_value)
            return value


fib = FibonacciSequence(30)
```

Now, `fib` is an actual iterator, it implements both `__iter__` and `__next__`. We can use it in a loop to get values, since we can call `__iter__` to get an iterator from it, and then loop over the elements.

```
for i in fib:
    print(i)
# But remember, when an there are no more elements to loop, the
```

```
# iterator is spent, and we need a new one to iterate again.
next(fib)

for i in FibonacciSequence(50):
    print(i)
```

Notice that in an iterator, we do not compute all of its elements at once. We obtain the elements as we go. This means, we can create iterators that store infinite numbers:

```
class NaturalNumbers:
    def __init__(self):
        self.n = 1

    def __iter__(self):
        return self

    def __next__(self):
        value = self.n
        self.n += 1
        return value


n = NaturalNumbers()
next(n)
```

We can call `n` forever, it is now a list with an inifinite number of elements.

The idea behind iterators is that you have a set of things you want to do, say very complicated tasks. But you would like to get the results as you finish each task. An iterator allows you to do just that. When we call `next`, it will do the first complicated task in the list, which might take awhile. When it is finished, it will return you the value so that you can analyze it or store it. We can then call `next` again to begin the next task, and so on.

8 Generators

Generators are simple iterators. Iterators are a very general idea, that require us to build a class and define `__iter__` and `__next__`. We gain a lot of control and can add a lot of complexity to an iterator. But, we do need to write more code for it.

Generators are a way to construct simple iterators. Instead of constructing a class, we will construct a function using the keyword `yield`. We use the keyword `yield` whenever we want to return a value and stop so that other code can execute. This is what was happening in the for-loop with an iterator, we called `next`, got some value back, ran the code inside the loop, and then called `next` again, and so on. Let's recreate the natural numbers example with generators:

```
class NaturalNumbers:
    def __init__(self):
        self.n = 1

    def __iter__(self):
        return self
```

```
    def __next__(self):
        value = self.n
        self.n += 1
        return value


def natural_numbers():
    n = 1
    while True:
        yield n
        n += 1


natural_generator = natural_numbers()
type(natural_generator)
# A generator behaves just like an iterator, but is created with
# the yield keyword.
# We can call next on the iterator to get the values:
next(natural_generator)
```

We can create a finite generator:

```
def names():
    yield "A"
    yield "B"
    yield "C"

names_gen = names()
# When we call next for the first time, the function will start
# being executed. When it finds the first yield, it returns that
# value and stops the execution of the function at that line.
# We can then do other things:
first_name = next(names_gen)
print(f"The first name is: {first_name}")
# We can then go back to that function again by calling next, which
# will continue the execution from where it started
second_name = next(names_gen)
print(f"The second name is: {second_name}")
# When we call next, we will reach the last yield, and after that,
# calling next would lead to StopIteration exceptions.
next(names_gen)
next(names_gen)
```

Generators are great to create generators (that are iterators) in a more readable fashion:

```
def fibonacci(n):
    """Get the first n elements of the Fibonacci sequence."""
    prev = [1]
    for i in range(n):
        yield prev[-1]
        prev.append(sum(prev[-2:]))
```

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

9 Generator Expressions

Generator expressions are a way to create generators, which are simple iterators, in an even simpler way. Generator expressions allow us to create generators in a single line:

```
def names_gen():
    for name in ["A", "B", "C"]:
        yield name

# equivalent to:
names_gen = (name for name in ["A", "B", "C"])
type(names_gen)
next(names_gen)
next(names_gen)
next(names_gen)
next(names_gen)
# raises StopIteration
```

We can use the built-in `list`, to get a list of values out of a generator

```
print(list(names_gen()))
```

We can also filter the values we give back in a generator expression:

```
even_numbers = (x for x in range(20) if x % 2 == 0)
print(list(even_numbers))

# and also modify numbers that we return
squared_number = (x**2 for x in range(10))
print(list(squared_number))
```

The full syntax for generator expressions is:

```
generator = (expression for value in collection if condition)
```

Avoid creating generator expressions that extend over a line. The syntax is meant for simple generators.

10 List Comprehensions

We have used list comprehensions in a few of the examples. List comprehensions are used to build simple lists, very much like generator expressions are used to build simple generators. The syntax of list comprehensions and generator expressions is basically the same, with the difference that list comprehensions use square-brackets, while generator expressions use parentheses.

```
# create a simple list of squared numbers
squared_numbers = [x**2 for x in range(10)]
print(squared_numbers)

# create a new list with filtered strings
```

```
messy_data = ['JOHN     ', ' JackSOn ', 'Sophia    ']
clean_data = [name.strip().capitalize() for name in messy_data]
```

List comprehensions create lists. Generator expressions create generators. Lists store all the values in memory at once, while generators create the values as requested.

11 Chaining Iterators

Iterators can be chained together to create an efficient workflow.

```
def naturals():
    n = 1
    while True:
        yield n
        n += 1


def squared(sequence):
    for i in sequence:
        yield i**2


chain = squared(naturals())
type(chain)
next(chain)
```

When we call `next` on chain, we start executing `squared`. When we reach the for-loop, the loop will call next on `sequence`. But `sequence` is our generator `naturals()`. Calling next on `naturals()` will start executing it, and it will yield the first number. This number is stored into the variable `i`, which is then squared and yielded to us.

```
def printed(sequence):
    for i in sequence:
        print(f"Number: {i}")
        yield i

chain = squared(printed(naturals()))
next(chain)
```

12 Context Managers

We used context managers before when dealing with files. The idea of context managers is that you want to execute some code, but you need something else done after the code is finished or even if the code fails. In the case of files, we need the file to be properly closed, so even if our code fails, the file closes and we do not run the risk of corrupting the file.

We start a context manager with the keyword `with`:

```
with open('my_file.txt', 'r') as f:
    for line in f:
        print(line)
```

```
# equivalent to
f = open('my_file.txt', 'r')
try:
    for line in f:
        print(line)
finally:
    f.close()
# Even if the code inside the try block leads to an error,
# the code inside the finally block will execute, properly
# closing the file.
```

The `open` function returns an object, as everything in Python is an object, that can be used as a context manager. An object that can be used as a context manager implements the `__enter__` and `__exit__` methods. Let's replicate the functionality of `open` with our own context manager:

```
class FileReader:
    def __init__(self, filename):
        self.filename = filename

    def __enter__(self):
        self.file = open(self.filename, 'r')
        return self.file

    def __exit__(self, *args):
        self.file.close()


with FileReader('my_file.txt') as f:
    for line in f:
        print(line)
```

When we call `FileReader` on the `with` line two things happen:

1. The class constructor is executed with '$\text{my}_{\text{file.txt}}$' as the input;
2. The `__enter__` method is executed and its return value is assigned to `f`.

Then we run our for-loop, and after the loop finishes, the `__exit__` method of the `FileReader` instance is executed, closing the file.

13 String Formatting

We will learn two ways of formatting strings in Python. First, using the `format` method of strings:

```
from datetime import date
name = "World"
print("Hello, {}!".format(name))

print("Hello, {}! Today's date is {}.".format(name, date.today()))
print("Hello, {name}! Today's date is {date}.".format(
    name=name, date=date.today()))
```

Second, is using formatted string literals:

```
print(f"Hello, {name}!")
print(f"Hello, {name}! Today's date is {date.today()}.")
print(f"5 + 2 = {5 + 2}")
from math import pi
print(f"Pi with 8 decimal cases: {pi:.8f}")
print(f"Pi with 4 decimal cases: {pi:.4f}")
# The colen specifies how to format the variable when
# constructing the string.
# .f means the number is a float
# .8f means we want to use 8 decimal places
```

Formatted string literals were introduced in Python 3.6, so it is not available on earlier Python versions. Formatting strings with `format` is available ever since the introduction of Python 3. For more details on the formatting options check the String Formatting reference page.

14 Debugging

Python has a built-in debugger, available in the module `pdb`. We can use `pdb` to insert breakpoints in the code, which interrupt the execution of the code and allow us to inspect the variables in the stack.

```
import pdb

def code_that_will_fail():
    x = [1, 2, 3]
    y = [3, 4, 5]
    pdb.set_trace()             # breakpoint
    z = x @ y
    return z


code_that_will_fail()
```

When the code hits the `pdb.set_trace`, it will interrupt execution, and you will get a message in the REPL that starts with `(Pdb)`. You can type code and hit enter to execute. For example, you can type `x` to inspect its value. You can then use the Debugger Commands to continue executing the code. You can continue the execution with:

- `next`: executes the next line
- `step`: executes the next line, and if it is a function, goes inside it and stops there (so you can see what is happening inside the function)
- `return`: executes all lines until a return is found
- `continue`: execute all lines until another breakpoint is found

You can also type `help` to get a list of the commands. To quit the debugger just type `exit`.

Another way to enter the debug mode is post-mortem, where the code failed and you did not set up any breakpoints. You can run `pdb.pm()` to debug the last error.

```
def code_that_will_fail():
    x = [1, 2, 3]
    y = [3, 4, 5]
    z = x @ y
    return z


code_that_will_fail()
pdb.pm()
```

This is useful for debugging smaller pieces of code.

15 Assignment

Problem 1 *Study the presentation: Understanding Ideas Behind Advanced Python Features. Summarize the main points of the talk.*