Python Programming for Data Science

Chapter 3: Unit Tests & Classes

Chapter Outline

Chapter Learning Objectives

- Formulate a test case to prove a function design specification.
- Use an `assert` statement to validate a test case.
- Debug Python code with the `pdb` module, or by using `%debug` in a Jupyter code cell.
- Describe the difference between a `class` and a `function` in Python.
- Be able to create a `class`.
- Differentiate between `instance attributes` and `class attributes`.
- Differentiate between `methods`, `class methods` and `static methods`.
- Understand and implement `subclassing`/`inheritance` with Python classes.

1. Unit Tests

Last chapter we discussed Python functions. But how can we be sure that our function is doing exactly what we expect it to do? **Unit testing** is the process of testing our function to ensure it's giving us the results we expect. Let's briefly introduce the concept here.

`assert` Statements

`assert` statements are the most common way to test your functions. They cause your program to fail if the tested condition is `False`. The syntax is:

```
assert expression, "Error message if expression is False or raises an error."
```

```
assert 1 == 2, "1 is not equal to 2."
```

```
---------------------------------------------------------------------
------
AssertionError                            Traceback (most recent call
last)
<ipython-input-1-74b71e52d7cf> in <module>
----> 1 assert 1 == 2, "1 is not equal to 2."

AssertionError: 1 is not equal to 2.
```

Asserting that two numbers are approximately equal can also be helpful. Due to the limitations of floating-point arithmetic in computers, numbers we expect to be equal are sometimes not:

```
assert 0.1 + 0.2 == 0.3, "Not equal!"
```

```
---------------------------------------------------------------------
------
AssertionError                            Traceback (most recent call
last)
<ipython-input-2-be33c16f53c8> in <module>
----> 1 assert 0.1 + 0.2 == 0.3, "Not equal!"

AssertionError: Not equal!
```

```
import math  # we'll learn about importing modules next chapter
assert math.isclose(0.1 + 0.2, 0.3), "Not equal!"
```

You can test any statement that evaluates to a boolean:

```
assert 'varada' in ['mike', 'tom', 'tiffany'], "Instructor not
present!"
```

```
---------------------------------------------------------------------
------
AssertionError                            Traceback (most recent call
last)
<ipython-input-4-ff5cfa3cbd54> in <module>
----> 1 assert 'varada' in ['mike', 'tom', 'tiffany'], "Instructor
not present!"

AssertionError: Instructor not present!
```

Test Driven Development

Test Driven Development (TDD) is where you write your tests before your actual function (more here). This may seem a little counter-intuitive, but you're creating the expectations of your function before the actual function. This can be helpful for several reasons:

- you will better understand exactly what code you need to write;
- you are forced to write tests upfront;
- you won't encounter large time-consuming bugs down the line; and,
- it helps to keep your workflow manageable by focussing on small, incremental code improvements and additions.

In general, the approach is as follows:

1. Write a stub: a function that does nothing but accept all input parameters and return the correct datatype.
2. Write tests to satisfy your design specifications.
3. Outline the program with pseudo-code.
4. Write code and test frequently.
5. Write documentation.

EAFP vs LBYL

Somewhat related to testing and function design are the philosophies EAFP and LBYL. EAFP = "Easier to ask for fogiveness than permission". In coding lingo: try doing something, and if it doesn't work, catch the error. LBYL = "Look before you leap". In coding lingo: check that you can do something before trying to do it. These two acronyms refer to coding philosophies about how to write your code. Let's see an example:

```
d = {'name': 'Doctor Python',
     'superpower': 'programming',
     'weakness': 'mountain dew',
     'enemies': 10}
```

```
# EAFP
try:
    d['address']
except KeyError:
    print('Please forgive me!')
```

```
Please forgive me!
```

```
# LBYL
if 'address' in d.keys():
    d['address']
else:
    print('Saved you before you leapt!')
```

```
Saved you before you leapt!
```

While EAFP is often vouched for in Python, there's no right and wrong way to code and it's often context-specific. I personally mix the two philosophies most of the time.

2. Debugging

So if your Python code doesn't work: what do you do? At the moment, most of you probably do "manual testing" or "exploratory testing". You keep changing your code until it works, maybe add some print statements around the place to isolate any problems. For example, consider the `random_walker` code below, which is adapted with permission from COS 126, Conditionals and Loops:

```
from random import random

def random_walker(T):
    """
    Simulates T steps of a 2D random walk, and prints the result after
each step.
    Returns the squared distance from the origin.

    Parameters
    ----------
    T : int
        The number of steps to take.

    Returns
    -------
    float
        The squared distance from the origin to the endpoint, rounded
to 2 decimals.

    Examples
    --------
    >>> random_walker(3)
    (0, -1)
    (0, 0)
    (0, -1)
    1.0
    """

    x = 0
    y = 0

    for i in range(T):
        rand = random()
        if rand < 0.25:
            x += 1
        if rand < 0.5:
            x -= 1
        if rand < 0.75:
            y += 1
        else:
            y -= 1
        print((x, y))
    return round((x ** 2 + y ** 2) ** 0.5, 2)

random_walker(5)
```

```
(0, -1)
(0, -2)
(-1, -1)
(-1, -2)
(-2, -1)
```

```
2.24
```

If we re-run the code above, our random walker never goes right (the x-coordinate is never +ve). We might try to add some print statements here to see what's going on:

```
def random_walker(T):
    """
    Simulates T steps of a 2D random walk, and prints the result after
each step.
    Returns the squared distance from the origin.

    Parameters
    ----------
    T : int
        The number of steps to take.

    Returns
    -------
    float
        The squared distance from the origin to the endpoint, rounded
to 2 decimals.

    Examples
    --------
    >>> random_walker(3)
    (0, -1)
    (0, 0)
    (0, -1)
    1.0
    """

    x = 0
    y = 0

    for i in range(T):
        rand = random()
        print(rand)
        if rand < 0.25:
            print("I'm going right!")
            x += 1
        if rand < 0.5:
            print("I'm going left!")
            x -= 1
        if rand < 0.75:
            y += 1
        else:
            y -= 1
        print((x, y))
    return round((x ** 2 + y ** 2) ** 0.5, 2)

random_walker(5)
```

```
0.7300233734078166
(0, 1)
0.25581301453119265
I'm going left!
(-1, 2)
0.06649453336551836
I'm going right!
I'm going left!
(-1, 3)
0.09637727201190915
I'm going right!
I'm going left!
(-1, 4)
0.06654466325447528
I'm going right!
I'm going left!
(-1, 5)
```

```
5.1
```

Ah! We see that even every time after a `"I'm going right!"` we immediately get a `"I'm going left!"`. The problem is in our `if` statements, we should be using `elif` for each statement after the intial `if`, otherwise multiple conditions may be met each time.

This was a pretty simple debugging case, adding print statements is not always helpful or efficient. Alternatively we can use the module `pdb`. `pdb` is the Python Debugger included with the standard library. We can use `breakpoint()` to leverage `pdb` and set a “break point” at any point in our code and then inspect our variables. See the `pdb` docs here and this cheatsheet for help interacting with the debugger console.

```
def random_walker(T):
    """
    Simulates T steps of a 2D random walk, and prints the result after
each step.
    Returns the squared distance from the origin.

    Parameters
    ----------
    T : int
        The number of steps to take.

    Returns
    -------
    float
        The squared distance from the origin to the endpoint, rounded
to 2 decimals.

    Examples
    --------
    >>> random_walker(3)
    (0, -1)
    (0, 0)
    (0, -1)
    1.0
    """

    x = 0
    y = 0

    for i in range(T):
        rand = random()
        breakpoint()
        if rand < 0.25:
            print("I'm going right!")
            x += 1
        if rand < 0.5:
            print("I'm going left!")
            x -= 1
        if rand < 0.75:
            y += 1
        else:
            y -= 1
        print((x, y))
    return round((x ** 2 + y ** 2) ** 0.5, 2)

random_walker(5)
```

```
> <ipython-input-10-005ed635a05e>(31)random_walker()
     29         rand = random()
     30         breakpoint()
---> 31         if rand < 0.25:
     32             print("I'm going right!")
     33             x += 1
```

```
---------------------------------------------------------------------------
------
StdinNotImplementedError                  Traceback (most recent call
last)
<ipython-input-10-005ed635a05e> in <module>
     42     return round((x ** 2 + y ** 2) ** 0.5, 2)
     43
---> 44 random_walker(5)

<ipython-input-10-005ed635a05e> in random_walker(T)
     29         rand = random()
     30         breakpoint()
---> 31         if rand < 0.25:
     32             print("I'm going right!")
     33             x += 1

<ipython-input-10-005ed635a05e> in random_walker(T)
     29         rand = random()
     30         breakpoint()
---> 31         if rand < 0.25:
     32             print("I'm going right!")
     33             x += 1

/opt/miniconda3/envs/py4ds/lib/python3.8/bdb.py in
trace_dispatch(self, frame, event, arg)
     86             return # None
     87         if event == 'line':
---> 88             return self.dispatch_line(frame)
     89         if event == 'call':
     90             return self.dispatch_call(frame, arg)

/opt/miniconda3/envs/py4ds/lib/python3.8/bdb.py in
dispatch_line(self, frame)
    110         """
    111         if self.stop_here(frame) or self.break_here(frame):
--> 112             self.user_line(frame)
    113             if self.quitting: raise BdbQuit
    114         return self.trace_dispatch

/opt/miniconda3/envs/py4ds/lib/python3.8/pdb.py in user_line(self,
frame)
    260             self._wait_for_mainpyfile = False
    261         if self.bp_commands(frame):
--> 262             self.interaction(frame, None)
    263
    264     def bp_commands(self, frame):

/opt/miniconda3/envs/py4ds/lib/python3.8/site-
packages/IPython/core/debugger.py in interaction(self, frame,
traceback)
    317     def interaction(self, frame, traceback):
    318         try:
--> 319             OldPdb.interaction(self, frame, traceback)
    320         except KeyboardInterrupt:
    321             self.stdout.write("\n" +
self.shell.get_exception_only())

/opt/miniconda3/envs/py4ds/lib/python3.8/pdb.py in interaction(self,
frame, traceback)
    355             return
    356         self.print_stack_entry(self.stack[self.curindex])
--> 357         self._cmdloop()
    358         self.forget()
    359

/opt/miniconda3/envs/py4ds/lib/python3.8/site-
packages/IPython/core/debugger.py in _cmdloop(self)
    817                 # the current command, so allow them during
interactive input
    818                 self.allow_kbdint = True
--> 819                 self.cmdloop()
    820                 self.allow_kbdint = False
    821                 break

/opt/miniconda3/envs/py4ds/lib/python3.8/site-
packages/IPython/core/debugger.py in cmdloop(self)
    803         """Wrap cmdloop() such that KeyboardInterrupt stops
the debugger."""
    804         try:
--> 805             return OldPdb.cmdloop(self)
    806         except KeyboardInterrupt:
    807             self.stop_here = lambda frame: False

/opt/miniconda3/envs/py4ds/lib/python3.8/cmd.py in cmdloop(self,
intro)
    124                     if self.use_rawinput:
    125                         try:
--> 126                             line = input(self.prompt)
```

```
    127                         except EOFError:
    128                             line = 'EOF'

/opt/miniconda3/envs/py4ds/lib/python3.8/site-
packages/ipykernel/kernelbase.py in raw_input(self, prompt)
    852         """
    853         if not self._allow_stdin:
--> 854             raise StdinNotImplementedError(
    855                 "raw_input was called, but this frontend does
not support input requests."
    856             )

StdinNotImplementedError: raw_input was called, but this frontend
does not support input requests.
```

So the correct code should be:

```
def random_walker(T):
    """
    Simulates T steps of a 2D random walk, and prints the result after
each step.
    Returns the squared distance from the origin.

    Parameters
    ----------
    T : int
        The number of steps to take.

    Returns
    -------
    float
        The squared distance from the origin to the endpoint, rounded
to 2 decimals.

    Examples
    --------
    >>> random_walker(3)
    (0, -1)
    (0, 0)
    (0, -1)
    1.0
    """

    x = 0
    y = 0

    for i in range(T):
        rand = random()
        if rand < 0.25:
            x += 1
        elif rand < 0.5:
            x -= 1
        elif rand < 0.75:
            y += 1
        else:
            y -= 1
        print((x, y))
    return round((x ** 2 + y ** 2) ** 0.5, 2)

random_walker(5)
```

```
(0, -1)
(-1, -1)
(-1, -2)
(-1, -1)
(-2, -1)
```

```
2.24
```

I wanted to show `pdb` because it's the standard Python debugger. Most Python IDE's also have their own debugging workflow, for example, here's a tutorial on debugging in VSCode. Within Jupyter, there is some "magic" commands that you can use. The one we are interested in here is `%debug`. There are a few ways you can use it, but the easiest is if a cell raises an error, we can create a new cell underneath and just write `%debug` and run that cell to debug our previous error.

```
x = 1
x + 'string'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-496b359b128e> in <module>
      1 x = 1
----> 2 x + 'string'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
%debug
```

```
> <ipython-input-12-496b359b128e>(2)<module>()
      1 x = 1
----> 2 x + 'string'
```

```
------------------------------------------------------------------------
------
StdinNotImplementedError                  Traceback (most recent call
last)
<ipython-input-13-bc99e4ec804d> in <module>
----> 1 get_ipython().run_line_magic('debug', '')

/opt/miniconda3/envs/py4ds/lib/python3.8/site-
packages/IPython/core/interactiveshell.py in run_line_magic(self,
magic_name, line, _stack_depth)
   2325                 kwargs['local_ns'] =
self.get_local_scope(stack_depth)
   2326             with self.builtin_trap:
-> 2327                 result = fn(*args, **kwargs)
   2328             return result
   2329

<decorator-gen-50> in debug(self, line, cell)

/opt/miniconda3/envs/py4ds/lib/python3.8/site-
packages/IPython/core/magic.py in <lambda>(f, *a, **k)
    185     # but it's overkill for just that one bit of state.
    186     def magic_deco(arg):
--> 187         call = lambda f, *a, **k: f(*a, **k)
    188
    189         if callable(arg):

/opt/miniconda3/envs/py4ds/lib/python3.8/site-
packages/IPython/core/magics/execution.py in debug(self, line, cell)
    467
    468         if not (args.breakpoint or args.statement or cell):
--> 469             self._debug_post_mortem()
    470         elif not (args.breakpoint or cell):
    471             # If there is no breakpoints, the line is just
code to execute

/opt/miniconda3/envs/py4ds/lib/python3.8/site-
packages/IPython/core/magics/execution.py in _debug_post_mortem(self)
    481
    482     def _debug_post_mortem(self):
--> 483         self.shell.debugger(force=True)
    484
    485     def _debug_exec(self, code, breakpoint):

/opt/miniconda3/envs/py4ds/lib/python3.8/site-
packages/IPython/core/interactiveshell.py in debugger(self, force)
   1190             return
   1191
-> 1192         self.InteractiveTB.debugger(force=True)
   1193
   1194     #-------------------------------------------------------
-----------------

/opt/miniconda3/envs/py4ds/lib/python3.8/site-
packages/IPython/core/ultratb.py in debugger(self, force)
   1265                     etb = etb.tb_next
   1266                 self.pdb.botframe = etb.tb_frame
-> 1267                 self.pdb.interaction(None, etb)
   1268
   1269         if hasattr(self, 'tb'):

/opt/miniconda3/envs/py4ds/lib/python3.8/site-
packages/IPython/core/debugger.py in interaction(self, frame,
traceback)
    317     def interaction(self, frame, traceback):
    318         try:
--> 319             OldPdb.interaction(self, frame, traceback)
    320         except KeyboardInterrupt:
    321             self.stdout.write("\n" +
self.shell.get_exception_only())

/opt/miniconda3/envs/py4ds/lib/python3.8/pdb.py in interaction(self,
frame, traceback)
    355             return
    356         self.print_stack_entry(self.stack[self.curindex])
--> 357         self._cmdloop()
    358         self.forget()
    359

/opt/miniconda3/envs/py4ds/lib/python3.8/pdb.py in _cmdloop(self)
    320                 # the current command, so allow them during
interactive input
    321                 self.allow_kbdint = True
--> 322                 self.cmdloop()
    323                 self.allow_kbdint = False
    324                 break

/opt/miniconda3/envs/py4ds/lib/python3.8/cmd.py in cmdloop(self,
```

```
intro)
    124                     if self.use_rawinput:
    125                         try:
--> 126                             line = input(self.prompt)
    127                         except EOFError:
    128                             line = 'EOF'

/opt/miniconda3/envs/py4ds/lib/python3.8/site-
packages/ipykernel/kernelbase.py in raw_input(self, prompt)
    852         """
    853         if not self._allow_stdin:
--> 854             raise StdinNotImplementedError(
    855                 "raw_input was called, but this frontend does
not support input requests."
    856             )

StdinNotImplementedError: raw_input was called, but this frontend
does not support input requests.
```

The JupyterLab variable inspector extension is another related helpful tool.

3. Python Classes

We've seen data types like `dict` and `list` which are built into Python. We can also create our own data types. These are called **classes** and an instance of a class is called an **object** (classes documentation here). The general approach to programming using classes and objects is called object-oriented programming

```
d = dict()
```

Here, `d` is an object, whereas `dict` is a type

```
type(d)
```

```
dict
```

```
type(dict)
```

```
type
```

We say `d` is an **instance** of the **type** `dict`. Hence:

```
isinstance(d, dict)
```

```
True
```

Why Create Your Own Types/Classes?

"Classes provide a means of bundling data and functionality together" (from the Python docs), in a way that's easy to use, reuse and build upon. It's easiest to discover the utility of classes through an example so let's get started!

Say we want to start storing information about students and instructors in the University of British Columbia's Master of Data Science Program (MDS).

> **Note**
>
> Recall that the content of this site is adapted from material I used to teach the 2020/2021 offering of the course "DSCI 511 Python Programming for Data Science" for the University of British Columbia's Master of Data Science Program.

We'll start with first name, last name, and email address in a dictionary:

```
mds_1 = {'first': 'Tom',
         'last': 'Beuzen',
         'email': 'tom.beuzen@mds.com'}
```

We also want to be able to extract a member's full name from their first and last name, but don't want to have to write out this information again. A function could be good for this:

```
def full_name(first, last):
    """Concatenate first and last with a space."""
    return f"{first} {last}"
```

```
full_name(mds_1['first'], mds_1['last'])
```

```
'Tom Beuzen'
```

We can just copy-paste the same code to create new members:

```
mds_2 = {'first': 'Tiffany',
         'last': 'Timbers',
         'email': 'tiffany.timbers@mds.com'}
full_name(mds_2['first'], mds_2['last'])
```

```
'Tiffany Timbers'
```

Creating a Class

The above was pretty inefficient. You can imagine that the more objects we want and the more complicated the objects get (more data, more functions) the worse this problem becomes! However, this is a perfect use case for a class! A class can be thought of as a **blueprint** for creating objects, in this case MDS members.

Terminology alert:

- Class data = "Attributes"
- Class functions = "Methods"

Syntax alert:

- We define a class with the `class` keyword, followed by a name and a colon (`:`):

```
class mds_member:
    pass
```

```
mds_1 = mds_member()
type(mds_1)
```

```
__main__.mds_member
```

We can add an `__init__` method to our class which will be run every time we create a new instance, for example, to add data to the instance. Let's add an `__init__` method to our `mds_member` class. `self` refers to the instance of a class and should always be passed to class methods as the first argument.

```
class mds_member:

    def __init__(self, first, last):
        # the below are called "attributes"
        self.first = first
        self.last = last
        self.email = first.lower() + "." + last.lower() + "@mds.com"
```

```
mds_1 = mds_member('Varada', 'Kolhatkar')
print(mds_1.first)
print(mds_1.last)
print(mds_1.email)
```

```
Varada
Kolhatkar
varada.kolhatkar@mds.com
```

To get the full name, we can use the function we defined earlier:

```
full_name(mds_1.first, mds_1.last)
```

```
'Varada Kolhatkar'
```

But a better way to do this is to integrate this function into our class as a `method`:

```
class mds_member:

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.email = first.lower() + "." + last.lower() + "@mds.com"

    def full_name(self):
        return f"{self.first} {self.last}"
```

```
mds_1 = mds_member('Varada', 'Kolhatkar')
print(mds_1.first)
print(mds_1.last)
print(mds_1.email)
print(mds_1.full_name())
```

```
Varada
Kolhatkar
varada.kolhatkar@mds.com
Varada Kolhatkar
```

Notice that we need the parentheses above because we are calling a `method` (think of it as a function), not an `attribute`.

Instance & Class Attributes

Attributes like `mds_1.first` are sometimes called `instance attributes`. They are specific to the object we have created. But we can also set `class attributes` which are the same amongst all instances of a class, they are defined outside of the `__init__` method.

```
class mds_member:

    role = "MDS member" # class attributes
    campus = "UBC"

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.email = first.lower() + "." + last.lower() + "@mds.com"

    def full_name(self):
        return f"{self.first} {self.last}"
```

All instances of our class share the class attribute:

```
mds_1 = mds_member('Tom', 'Beuzen')
mds_2 = mds_member('Joel', 'Ostblom')
print(f"{mds_1.first} is at campus {mds_1.campus}.")
print(f"{mds_2.first} is at campus {mds_2.campus}.")
```

```
Tom is at campus UBC.
Joel is at campus UBC.
```

We can even change the class attribute after our instances have been created. This will affect all of our created instances:

```
mds_1 = mds_member('Tom', 'Beuzen')
mds_2 = mds_member('Mike', 'Gelbart')
mds_member.campus = 'UBC Okanagan'

print(f"{mds_1.first} is at campus {mds_1.campus}.")
print(f"{mds_2.first} is at campus {mds_2.campus}.")
```

```
Tom is at campus UBC Okanagan.
Mike is at campus UBC Okanagan.
```

You can also change the class attribute for just a single instance. But this is typically not recommended because if you want differing attributes for instances, you should probably use `instance attributes`.

```
class mds_member:

    role = "MDS member"
    campus = "UBC"

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.email = first.lower() + "." + last.lower() + "@mds.com"

    def full_name(self):
        return f"{self.first} {self.last}"
```

```
mds_1 = mds_member('Tom', 'Beuzen')
mds_2 = mds_member('Mike', 'Gelbart')
mds_1.campus = 'UBC Okanagan'

print(f"{mds_1.first} is at campus {mds_1.campus}.")
print(f"{mds_2.first} is at campus {mds_2.campus}.")
```

```
Tom is at campus UBC Okanagan.
Mike is at campus UBC.
```

Methods, Class Methods & Static Methods

The `methods` we've seen so far are sometimes calles "regular" `methods`, they act on an instance of the class (i.e., take `self` as an argument). We also have `class methods` that act on the actual class. `class methods` are often used as "alternative constructors". As an example, let's say that somebody commonly wants to use our class with comma-separated names like the following:

```
name = 'Tom,Beuzen'
```

Unfortunately, those users can't do this:

```
mds_member(name)
```

```
---------------------------------------------------------------------
------
TypeError                                 Traceback (most recent call
last)
<ipython-input-35-aba2b627390d> in <module>
----> 1 mds_member(name)

TypeError: __init__() missing 1 required positional argument: 'last'
```

To use our class, they would need to parse this string into `first` and `last`:

```
first, last = name.split(',')
print(first)
print(last)
```

```
Tom
Beuzen
```

Then they could make an instance of our class:

```
mds_1 = mds_member(first, last)
```

If this is a common use case for the users of our code, we don't want them to have to coerce the data every time before using our class. Instead, we can facilitate their use-case with a `class method`. There are two things we need to do to use a `class method`:

1. Identify our method as `class method` using the decorator `@classmethod` (more on decorators in a bit);
2. Pass `cls` instead of `self` as the first argument.

```
class mds_member:

    role = "MDS member"
    campus = "UBC"

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.email = first.lower() + "." + last.lower() + "@mds.com"

    def full_name(self):
        return f"{self.first} {self.last}"

    @classmethod
    def from_csv(cls, csv_name):
        first, last = csv_name.split(',')
        return cls(first, last)
```

Now we can use our comma-separated values directly!

```
mds_1 = mds_member.from_csv('Tom,Beuzen')
mds_1.full_name()
```

```
'Tom Beuzen'
```

There is a third kind of method called a `static method`. `static methods` do not operate on either the instance or the class, they are just simple functions. But we might want to include them in our class because they are somehow related to our class. They are defined using the `@staticmethod` decorator:

```
class mds_member:

    role = "MDS member"
    campus = "UBC"

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.email = first.lower() + "." + last.lower() + "@mds.com"

    def full_name(self):
        return f"{self.first} {self.last}"

    @classmethod
    def from_csv(cls, csv_name):
        first, last = csv_name.split(',')
        return cls(first, last)

    @staticmethod
    def is_quizweek(week):
        return True if week in [3, 5] else False
```

Note that the method `is_quizweek()` does not accept or use the `self` argument. But it is still MDS-related, so we might want to include it here.

```
mds_1 = mds_member.from_csv('Tom,Beuzen')
print(f"Is week 1 a quiz week? {mds_1.is_quizweek(1)}")
print(f"Is week 3 a quiz week? {mds_1.is_quizweek(3)}")
```

```
Is week 1 a quiz week? False
Is week 3 a quiz week? True
```

Decorators

Decorators can be quite a complex topic, you can read more about them here. Briefly, they are what they sounds like, they "decorate" functions/methods with additional functionality. You can think of a decorator as a function that takes another function and adds functionality.

Let's create a decorator as an example. Recall that functions are data types in Python, they can be passed to other functions. So a decorator simply takes a function as an argument, adds some more functionality to it, and returns a "decorated function" that can be executed.

```
# some function we wish to decorate
def original_func():
    print("I'm the original function!")

# a decorator
def my_decorator(original_func):  # takes our original function as
input

    def wrapper():  # wraps our original function with some extra
functionality
        print(f"A decoration before {original_func.__name__}.")
        result = original_func()
        print(f"A decoration after {original_func.__name__}.")
        return result

    return wrapper  # returns the unexecuted wrapper function which we
can can excute later
```

The `my_decorator()` function will return to us a function which is the decorated version of our original function.

```
my_decorator(original_func)
```

```
<function __main__.my_decorator.<locals>.wrapper()>
```

As a function was returned to us, we can execute it by adding parentheses:

```
my_decorator(original_func)()
```

```
A decoration before original_func.
I'm the original function!
A decoration after original_func.
```

We can decorate any arbitrary function with our decorator:

```
def another_func():
    print("I'm a different function!")

my_decorator(another_func)()
```

```
A decoration before another_func.
I'm a different function!
A decoration after another_func.
```

The syntax of calling our decorator is not that readable. Instead, we can use the `@` symbol as “syntactic sugar” to improve readability and reuseability of decorators:

```
@my_decorator
def one_more_func():
    print("One more function...")

one_more_func()
```

```
A decoration before one_more_func.
One more function...
A decoration after one_more_func.
```

Okay, let’s make something a little more useful. We will create a decorator that times the execution time of any arbitrary function:

```
import time  # import the time module, we'll learn about imports next
chapter

def timer(my_function):  # the decorator

    def wrapper():  # the added functionality
        t1 = time.time()
        result = my_function()  # the original function
        t2 = time.time()
        print(f"{my_function.__name__} ran in {t2 - t1:.3f} sec")  #
print the execution time
        return result
    return wrapper
```

```
@timer
def silly_function():
    for i in range(10_000_000):
        if (i % 1_000_000) == 0:
            print(i)
        else:
            pass

silly_function()
```

```
0
```

```
1000000
2000000
```

```
3000000
```

```
4000000
```

```
5000000
6000000
```

```
7000000
```

```
8000000
9000000
```

```
silly_function ran in 0.570 sec
```

Python's built-in decorators like `classmethod` and `staticmethod` are coded in C so I'm not showing them here. I don't often create my own decorators, but I use the built-in decorators all the time.

Inheritance & Subclasses

Just like it sounds, inheritance allows us to "inherit" methods and attributes from another class. So far, we've been working with an `mds_member` class. But let's get more specific and create a `mds_student` and `mds_instructor` class. Recall this was `mds_member`:

```
class mds_member:

    role = "MDS member"
    campus = "UBC"

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.email = first.lower() + "." + last.lower() + "@mds.com"

    def full_name(self):
        return f"{self.first} {self.last}"

    @classmethod
    def from_csv(cls, csv):
        first, last = csv_name.split(',')
        return cls(first, last)

    @staticmethod
    def is_quizweek(week):
        return True if week in [3, 5] else False
```

We can create an `mds_student` class that inherits all of the attributes and methods from our `mds_member` class by by simply passing the `mds_member` class as an argument to an `mds_student` class definition:

```
class mds_student(mds_member):
    pass
```

```
student_1 = mds_student('Craig', 'Smith')
student_2 = mds_student('Megan', 'Scott')
print(student_1.full_name())
print(student_2.full_name())
```

```
Craig Smith
Megan Scott
```

What happened here is that our `mds_student` instance first looked in the `mds_student` class for an `__init__` method, which it didn't find. It then looked for the `__init__` method in the inherited `mds_member` class and found something to use! This order is called the "method resolution order". We can inspect it directly using the `help()` function:

```
help(mds_student)
```

```
Help on class mds_student in module __main__:

class mds_student(mds_member)
 |  mds_student(first, last)
 |
 |  Method resolution order:
 |      mds_student
 |      mds_member
 |      builtins.object
 |
 |  Methods inherited from mds_member:
 |
 |  __init__(self, first, last)
 |      Initialize self.  See help(type(self)) for accurate
signature.
 |
 |  full_name(self)
 |
 |  ----------------------------------------------------------------
-----
 |  Class methods inherited from mds_member:
 |
 |  from_csv(csv) from builtins.type
 |
 |  ----------------------------------------------------------------
-----
 |  Static methods inherited from mds_member:
 |
 |  is_quizweek(week)
 |
 |  ----------------------------------------------------------------
-----
 |  Data descriptors inherited from mds_member:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------
-----
 |  Data and other attributes inherited from mds_member:
 |
 |  campus = 'UBC'
 |
 |  role = 'MDS member'
```

Okay, let's fine-tune our `mds_student` class. The first thing we might want to do is change the role of the student instances to "MDS Student". We can do that by simply adding a `class attribute` to our `mds_student` class. Any attributes or methods not "over-ridden" in the `mds_student` class will just be inherited from the `mds_member` class.

```
class mds_student(mds_member):
    role = "MDS student"
```

```
student_1 = mds_student('John', 'Smith')
print(student_1.role)
print(student_1.campus)
print(student_1.full_name())
```

```
MDS student
UBC
John Smith
```

Now let's add an `instance attribute` to our class called `grade`. You might be tempted to do something like this:

```
class mds_student(mds_member):
    role = "MDS student"

    def __init__(self, first, last, grade):
        self.first = first
        self.last = last
        self.email = first.lower() + "." + last.lower() + "@mds.com"
        self.grade = grade

student_1 = mds_student('John', 'Smith', 'B+')
print(student_1.email)
print(student_1.grade)
```

```
john.smith@mds.com
B+
```

But this is not DRY code, remember that we've already typed most of this in our `mds_member` class. So what we can do is let the `mds_member` class handle our `first` and `last` argument and we'll just worry about `grade`. We can do this easily with the `super()` function. Things can get pretty complicated with `super()`, you can read more here, but all you really need to know is that `super()` allows you to inherit attributes/methods from other classes.

```
class mds_student(mds_member):
    role = "MDS student"

    def __init__(self, first, last, grade):
        super().__init__(first, last)
        self.grade = grade

student_1 = mds_student('John', 'Smith', 'B+')
print(student_1.email)
print(student_1.grade)
```

```
john.smith@mds.com
B+
```

Amazing! Hopefully you can start to see how powerful inheritance can be. Let's create another subclass called `mds_instructor`, which has two new methods `add_course()` and `remove_course()`.

```
class mds_instructor(mds_member):
    role = "MDS instructor"

    def __init__(self, first, last, courses=None):
        super().__init__(first, last)
        self.courses = ([] if courses is None else courses)

    def add_course(self, course):
        self.courses.append(course)

    def remove_course(self, course):
        self.courses.remove(course)
```

```
instructor_1 = mds_instructor('Tom', 'Beuzen', ['511', '561', '513'])
print(instructor_1.full_name())
print(instructor_1.courses)
```

```
Tom Beuzen
['511', '561', '513']
```

```
instructor_1.add_course('591')
instructor_1.remove_course('513')
instructor_1.courses
```

```
['511', '561', '591']
```

Getters/Setters/Deleters

There's one more import topic to talk about with Python classes and that is getters/setters/deleters. The necessity for these actions is best illustrated by example. Here's a stripped down version of the `mds_member` class from earlier:

```
class mds_member:

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.email = first.lower() + "." + last.lower() + "@mds.com"

    def full_name(self):
        return f"{self.first} {self.last}"
```

```
mds_1 = mds_member('Tom', 'Beuzen')
print(mds_1.first)
print(mds_1.last)
print(mds_1.email)
print(mds_1.full_name())
```

```
Tom
Beuzen
tom.beuzen@mds.com
Tom Beuzen
```

Imagine that I mis-spelled the name of this class instance and wanted to correct it. Watch what happens...

```
mds_1.first = 'Tomas'
print(mds_1.first)
print(mds_1.last)
print(mds_1.email)
print(mds_1.full_name())
```

```
Tomas
Beuzen
tom.beuzen@mds.com
Tomas Beuzen
```

Uh oh... the email didn't update with the new first name! We didn't have this problem with the `full_name()` method because it just calls the current `first` and `last` name. You might think that the best thing to do here is to create a method for `email()` like we have for `full_name()`. But this is bad coding for a variety of reasons, for example it means that users of your code will have to change every call to the `email` attribute to a call to the `email()` method. We'd call that a breaking change to our software and we want to avoid that where possible. What we can do instead, is define our `email` like a method, but keep it as an attribute using the `@property` decorator.

```
class mds_member:

    def __init__(self, first, last):
        self.first = first
        self.last = last

    def full_name(self):
        return f"{self.first} {self.last}"

    @property
    def email(self):
        return self.first.lower() + "." + self.last.lower() +
"@mds.com"
```

```
mds_1 = mds_member('Tom', 'Beuzen')
mds_1.first = 'Tomas'
print(mds_1.first)
print(mds_1.last)
print(mds_1.email)
print(mds_1.full_name())
```

```
Tomas
Beuzen
tomas.beuzen@mds.com
Tomas Beuzen
```

We could do the same with the `full_name()` method if we wanted too...

```
class mds_member:

    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def full_name(self):
        return f"{self.first} {self.last}"

    @property
    def email(self):
        return self.first.lower() + "." + self.last.lower() +
"@mds.com"
```

```
mds_1 = mds_member('Tom', 'Beuzen')
mds_1.full_name
```

```
'Tom Beuzen'
```

But what happens if we instead want to make a change to the full name now?

```
mds_1.full_name = 'Thomas Beuzen'
```

```
---------------------------------------------------------------------
------
AttributeError                            Traceback (most recent call
last)
<ipython-input-67-74e4ce79e805> in <module>
----> 1 mds_1.full_name = 'Thomas Beuzen'

AttributeError: can't set attribute
```

We get an error... Our class instance doesn't know what to do with the value it was passed. Ideally, we'd like our class instance to use this full name information to update `self.first` and `self.last`. To handle this action, we need a `setter`, defined using the decorator `@<attribute>.setter`:

```
class mds_member:

    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def full_name(self):
        return f"{self.first} {self.last}"

    @full_name.setter
    def full_name(self, name):
        first, last = name.split(' ')
        self.first = first
        self.last = last

    @property
    def email(self):
        return self.first.lower() + "." + self.last.lower() +
"@mds.com"
```

```
mds_1 = mds_member('Tom', 'Beuzen')
mds_1.full_name = 'Thomas Beuzen'
print(mds_1.first)
print(mds_1.last)
print(mds_1.email)
print(mds_1.full_name)
```

```
Thomas
Beuzen
thomas.beuzen@mds.com
Thomas Beuzen
```

Almost there! We've talked about getting information and setting information, but what about deletting information? This is typically used to do some clean up and is defined with the `@<attribute>.deleter` decorator. I rarely use this method but I want you to see it:

```
class mds_member:

    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def full_name(self):
        return f"{self.first} {self.last}"

    @full_name.setter
    def full_name(self, name):
        first, last = name.split(' ')
        self.first = first
        self.last = last

    @full_name.deleter
    def full_name(self):
        print('Name deleted!')
        self.first = None
        self.last = None

    @property
    def email(self):
        return self.first.lower() + "." + self.last.lower() +
"@mds.com"
```

```
mds_1 = mds_member('Tom', 'Beuzen')
delattr(mds_1, "full_name")
print(mds_1.first)
print(mds_1.last)
```

```
Name deleted!
None
None
```

Congrats for making it to the end, that was a lot of content and some tough topics to get through, so well done!!

By Tomas Beuzen