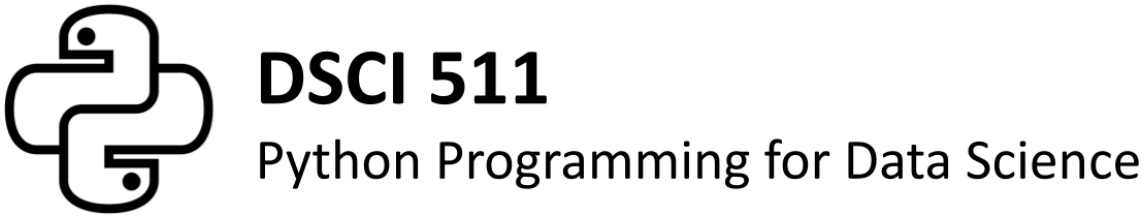
Lecture 8: Classes and Style

Contents



Lecture learning objectives

- 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
- Describe why code style is important

- Differentiate between the role of a linter like flake8 and an autoformatter like black
- Implement linting and formatting from the command line or within Jupyter or another IDE

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

Python Classes

We have encountered many kinds of objects in our first few weeks learning Python and Data Science. Some examples include the `ndarray` from numpy and the `Series` and `DataFrame` from pandas. Each of these "types of object" came with their own set of attributes and methods.

```
ser1 = pd.Series(['Vancouver', 'Edmonton', 'Toronto'])

# add a dot and hit TAB in jupyter-lab to see methods for this Series
```

```
# ser1.
```

```
ser2 = pd.Series([34.2, 30.3, 25.5])

#ser2.
```

We see the same attributes and methods for `ser1` and `ser2`, because they are both examples of the same "kind" of object in Python.

```

```

```
ar1 = np.array([34.2, 30.3, 25.5])

# ar1.
```

However, we notice that `ar1` has a different set of methods and attributes since it is an `ndarray` and not a pandas `Series`.

In Python, we say `ar1` is an **instance** of the `numpy.ndarray` **class**. Hence:

```
isinstance(ar1, np.ndarray)
```

```
True
```

```
isinstance(ser1, pd.Series)
```

```
True
```

Terminology

- **Class**: A class is a blueprint or template for creating objects. It defines the attributes and methods that an object will have, but does not create any objects itself. (e.g. `list`)

- **Object**: An object is an instance of a class. It has its own state (attributes) and behavior (methods), which are defined by the class. (e.g., `numbers = [1, 2, 3]`)

- **Method**: A method is a function that is associated with an object. It defines the behavior of the object, and can access and modify the object's attributes. (e.g. `numbers.append(4)`)

- **Attribute**: An attribute is a variable that holds data associated with an object. It defines the state of the object, and can be accessed and modified by the object's methods. (e.g., the values 1, 2, 3 are called the attributes of object `numbers`)

All along in DSCI 511, we have been organizing our code around objects whose attributes describe their current state and whose methods modify or update this state. This programming style is called “Object-oriented Programming” (OOP for short) and it is the manner in which most Python programs are written.

Python also allows us to define our own Classes to suit our needs. Today, we will learn how to create a Python Class, and how to instantiate an *object* that belongs to that Class (i.e. is created using that specific Class’s blueprint).

Why object-oriented programming is relevant in data science?

As we mentioned above, OOP is the most common paradigm for writing Python programs. In particular, many Python packages used in Data Science are written in an object-oriented manner, and you will need to understand the language of classes and objects in Python to use them.

For example, let’s say you are interested in building a deep learning model (e.g. a Convolution Neural Networks) to classify images using the PyTorch library. You start researching for a tutorial on how to build a CNN in PyTorch and you came across the following code:

```
class Net(nn.Module):
    def __init__(self):
      super(Net, self).__init__()

      self.conv1 = nn.Conv2d(1, 32, 3, 1)
      self.conv2 = nn.Conv2d(32, 64, 3, 1)

      self.dropout1 = nn.Dropout2d(0.25)
      self.dropout2 = nn.Dropout2d(0.5)

      self.fc1 = nn.Linear(9216, 128)
      self.fc2 = nn.Linear(128, 10)

my_nn = Net()
```

If you haven't learned about OOP before, this code might not make much sense to you. You might ask:

- What is `class Net(nn.Module)`?
- What is `__init__`?
- What is `super`?
- What is `self` and why did it appear multiple times throughout the code?

Without a proper understanding of OOP, you might struggle to define your own neural network architecture in Pytorch.

Many machine learning models that you come across are also written in OOP, for example: take a look at the source code of `RandomForestClassifer` in scikit-learn.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=1000, n_features=4,

clf = RandomForestClassifier(max_depth=2, random_state=0)
clf.fit(X, y)
clf.predict([[0, 0, 0, 0]])
```

Why create your own types/classes?

From the Python docs: "*Classes provide a means of bundling data and functionality together*". In particular, they help us bundle this functionality in a way that's easy to use, reuse and build upon. Let's get started with an example!

Say we want to start storing information about MDS students and instructors. We'll start with first name, last name, and initials in a dictionary:

```
mds_1 = {'first': 'Prajeet',
         'last': 'Bajpai',
         'initials': 'PB'}
```

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'])
```

```
'Prajeet Bajpai'
```

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

```
mds_2 = {'first': 'Tiffany',
         'last': 'Timbers',
         'initials': 'TT'}

full_name(mds_2['first'], mds_2['last'])
```

```
'Tiffany Timbers'
```

Creating a class

That 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! This is a good use case for defining a `Class`.

Define a class

To define a class, you use the `class` keyword followed by the name of the class and a colon (`:`). For example

```
class MdsMember:
    pass
```

In this example, we define a class called `MdsMember` that doesn't have any attributes or methods. The `pass` keyword is used to indicate that the class is empty

Creating an Object

To create an object from a class, you call the class like a function, passing any required arguments. For example

```
mds_1 = MdsMember()
print(type(mds_1))
```

```
<class '__main__.MdsMember'>
```

Adding attributes

We can use the `__init__` method to initialize the attributes with values passed to the arguments, which will be run every time we create a new instance, for example, to add data to the instance. This is called a **class constructor** - it is a special method used to initialize objects of a class.

Let's add an `__init__` method to our `MdsMember` class.

Note: `self` refers to the instance of a class and should always be passed to class method **definitions** as the first argument.

```
class MdsMember:

    def __init__(self, first, last):
        # here are the "attributes"
        self.first = first
        self.last = last
        self.initials = (first[0] + last[0]).upper()
```

```

```

```
mds_1 = MdsMember('Varada', 'Kolhatkar')
```

```
mds_1.first
```

```
'Varada'
```

```
mds_1.last
```

```
'Kolhatkar'
```

```
mds_1.initials
```

```
'VK'
```

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`.

Adding methods

To add methods to a class, you define them in the class definition.

```
class MdsMember:

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.initials = (first[0] + last[0]).upper()

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

We created a method called `full_name`. The methods takes no arguments, but uses the `self` parameter to access the object's attributes.

```
mds_1 = MdsMember('Varada', 'Kolhatkar')
```

```
mds_1.first
```

```
'Varada'
```

```
mds_1.last
```

```
'Kolhatkar'
```

```

```

```
mds_1.initials
```

```
'VK'
```

```
mds_1.full_name()
```

```
'Varada Kolhatkar'
```

NOTE: Notice that we need the parentheses above because we are calling a `method` (i.e., a function), not an `attribute`

Instance and class attributes

Attributes like `mds_1.first` are sometimes called **instance attributes**. They are specific to the object we have created

However, we can also set **class variables** which are the same amongst all instances of a class, they are defined outside of the `__init__` method

```

```

```
class MdsMember:

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

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.initials = (first[0] + last[0]).upper()

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

All instances of our class share the class variable:

```
mds_1 = MdsMember('Varada', 'Kolhatkar')
mds_2 = MdsMember('Joel', 'Ostblom')
```

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

```
Varada is at campus UBC.
Joel is at campus UBC.
```

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

```
mds_1 = MdsMember('Varada', 'Kolhatkar')
mds_2 = MdsMember('Joel', 'Ostblom')
MdsMember.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}.")
```

```
Varada is at campus UBC Okanagan.
Joel is at campus UBC Okanagan.
```

You can technically change a class variable for just a single instance, but this is typically not recommended. After all, if you want differing attributes for instances, you should probably use **instance attributes**!

```
mds_1.campus = 'UBC Vancouver'
```

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

```
Varada is at campus UBC Vancouver.
Joel is at campus UBC Okanagan.
```

Instance, class, and static methods

The methods we've seen so far are sometimes calls "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.

These **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 = 'Tiffany,Timbers'
```

- Unfortunately, those users can't do this:

```
MdsMember(name)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[35], line 1
----> 1 MdsMember(name)

TypeError: MdsMember.__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(',')
```

```
first
```

```
'Tiffany'
```

```
last
```

```
'Timbers'
```

Then they can make an instance of our class.

```
mds_1 = MdsMember(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:

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

```
class MdsMember:

    role = "MDS member"
    campus = "UBC"

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.initials = (first[0] + last[0]).upper()

    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 = MdsMember.from_csv('Scott,Mackie')
mds_1.full_name()
```

```
'Scott Mackie'
```

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. When you call a static method, no instance is passed as the first argument to the method. Nevertheless, we might want to include them in our class because they are somehow related to the class.

Static methods are defined using the `@staticmethod` decorator.

```
class MdsMember:

    role = "MDS member"
    campus = "UBC"

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.initials = (first[0] + last[0]).upper()

    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 = MdsMember.from_csv('Quan,Nguyen')
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
```

The key difference between regular methods and static methods is that regular methods can access and modiy the attributes of the instance they are called on, while static methods cannot. Thus static methods are typically used for utility functions that don't need to access or modify instance attributes.

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 `MdsMember` class. Now, let's get more specific and create a `MdsStudent` and `MdsInstructor` class. Recall what `MdsMember` looked like:

```
class MdsMember:

    role = "MDS member"
    campus = "UBC"

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.initials = (first[0] + last[0]).upper()

    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
```

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

```
class MdsStudent(MdsMember):
    pass
```

```
student_1 = MdsStudent('Craig', 'Smith')
student_2 = MdsStudent('Megan', 'Scott')
```

```
student_1.full_name()
```

```
'Craig Smith'
```

```
student_2.full_name()
```

```
'Megan Scott'
```

We call `MdsStudent` a **child** class of the **parent** `MdsMember` class.

What happened here is that our `MdsStudent` instance first looked in the `MdsStudent` class for an `__init__` method, which it didn't find. It then looked for the `__init__` method in the inherited `MdsMember` class and found something to use! This order is called the "method resolution order".

We can inspect the class directly using the `help()` function.

```
help(MdsStudent)
```

```
Help on class MdsStudent in module __main__:

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

```
|
|   role = 'MDS member'
```

Okay, let's fine-tune our `MdsStudent` 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 `MdsStudent` class. Any attributes or methods not "over-ridden" in the `MdsStudent` class will just be inherited from the `MdsMember` class.

```
class MdsStudent(MdsMember):
    role = "MDS student"
```

```
student_1 = MdsStudent('John', 'Smith')
```

```
student_1.role
```

```
'MDS student'
```

```
student_1.campus
```

```
'UBC'
```

```
student_1.full_name()
```

```
'John Smith'
```

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

```
class MdsStudent(MdsMember):
    role = "MDS student"

    def __init__(self, first, last, grade):
        self.first = first
        self.last = last
        self.initials = (first[0] + last[0]).upper()
        self.grade = grade
```

```
student_1 = MdsStudent('John', 'Smith', 'B+')
```

```
student_1.initials
```

```
'JS'
```

```
student_1.grade
```

```
'B+'
```

But this is not DRY code, remember that we've already typed most of this in our `MdsMember` class! So what we can do is let the `MdsMember` class handle our `first` and `last` argument and we'll just worry about `grade`

We can do this easily with the `super()` function.

Read more about `super()` here. Things can get complicated here, but what you most need to know is that `super()` allows you to inherit attributes/methods from other classes.

```
class MdsStudent(MdsMember):
    role = "MDS student"

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

```
student_1 = MdsStudent('John', 'Smith', 'B+')
```

```
student_1.initials
```

```
'JS'
```

```
student_1.grade
```

```
'B+'
```

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

```

```

```
class MdsInstructor(MdsMember):
    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 = MdsInstructor('Prajeet', 'Bajpai', ['511', '571', '591'])
```

```
instructor_1.full_name()
```

```
'Prajeet Bajpai'
```

```
instructor_1.courses
```

```
['511', '571', '591']
```

```
instructor_1.add_course('572')
instructor_1.remove_course('591')
```

```
instructor_1.courses
```

```
['511', '571', '572']
```

Magic Methods

Python also offers "magic methods" that allow you to use some basic, built-in operators for your own classes. For example, we can define how the Python function `len()` is interpreted by our `MdsInstructor` Class

```
class MdsInstructor(MdsMember):
    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)

    def __len__(self):
        return len(self.courses)
```

```
instructor_1 = MdsInstructor('Prajeet', 'Bajpai', ['511', '571', '591'])
len(instructor_1)
```

```
3
```

Magic methods are always accompanied by the "double underscore" in the name. There are several magic methods you can use to define the meaning of comparisons like `==`, `<` and `>`.

```
instructor_1 = MdsInstructor('Prajeet', 'Bajpai', ['511', '571', '591'])
instructor_2 = MdsInstructor('Prajeet', 'Bajpai', ['511', '571', '591'])
instructor_1 == instructor_2
```

```
False
```

```
class MdsInstructor(MdsMember):
    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)

    def __len__(self):
        return len(self.courses)

    def __eq__(self, other):
        return (self.first == other.first) and (self.last == other.last)
```

```
instructor_1 = MdsInstructor('Prajeet', 'Bajpai', ['511', '571', '591'])
instructor_2 = MdsInstructor('Prajeet', 'Bajpai', ['511', '571', '591'])
instructor_1 == instructor_2
```

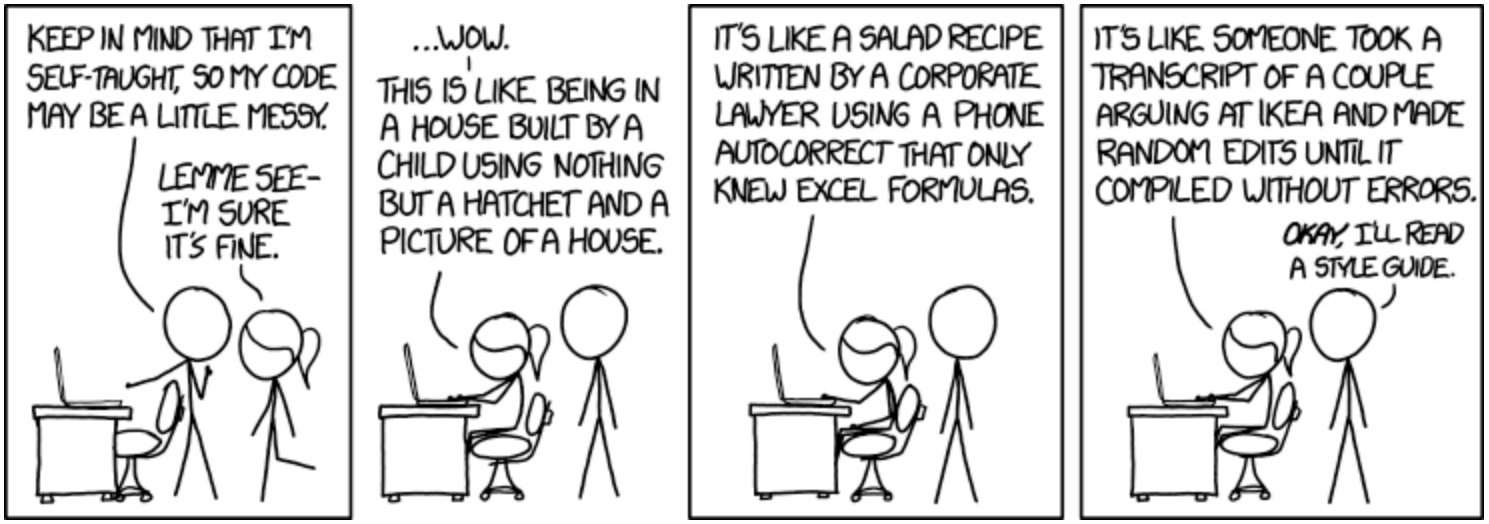
```
True
```

Be sure you only define `==` in a way that makes sense for your use case!

```
instructor_1 = MdsInstructor('Prajeet', 'Bajpai', ['511'])
instructor_2 = MdsInstructor('Prajeet', 'Bajpai', ['561'])
instructor_1 == instructor_2
```

```
True
```

Style Guides



It is incorrect to think that if your code works then you are done. Code has two "users": **the computer** (which turns it into machine instructions) and **humans** (who will likely read and/or modify the code in the future).

This section is about how to make your code suitable to that second audience, humans. Styling is particularly important for sharing your code to other users– including your future self! Remember: "Code is read much more often than it is written."

At MDS, we ask for the Python code you write to conform to the PEP 8 style guide.

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Highlights of PEP 8

Naming conventions:

For **variables, functions, modules, packages**: use snake_case i.e. lowercase words separated by underscores

```
# bad example
CarBrand = 'Toyota'
CARBRAND = 'Toyota'

# good example
car_brand = 'Toyota'
```

For **Classes**: use *camel case*, no underscores: `MyClass`, `Regressor`, `ErrorHandler`

```
# bad example
class my_birthday():
    pass

# good example
class MyBirthday():
    pass
```

Line breaking:

PEP 8 suggests 79 characters per line. Lines can be broken when code is contained within `()`, `[]`, `{}`:

```
# bad example
def my_function(a, b, c, d, e, f, g, h, i, j, k, l, m, n):
    return None

# good example
def my_function(a, b, c, d,
                e, f, g, h,
                i, j ,k, k,
                m, n):
    return None
```

> Note: lines can also be broken by `\`, but it is NOT recommended.

Lines should be broken before binary operators (e.g. `+` or `*`) not after:

```
# bad example
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

```
# good example
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Closing parentheses, brackets, or braces:

The closing brace/bracket/parenthesis on multiline constructs may either line up under the first non-whitespace character of the last line of list

```
# bad example
natural_numbers = [1, 2, 3,
                   4, 5, 6,
                   7, 8, 9]
# good example
natural_numbers = [
    1, 2, 3,
    4, 5, 6,
    7, 8, 9
]

# good example

natural_numbers = [
    1, 2, 3,
    4, 5, 6,
    7, 8, 9
    ]
```

Comments

- Comments are important for understanding your code.
- While docstrings cover what a function *does*, your comments will help document *how* your code achieves its goal.

`#` of an inline comment should be preceded with 2 spaces, and followed by 1 space. Use inline comments sparingly. Don't explain the obvious.

```
y = 100  # this is an inline comment
```

Block Comments: each line of a block comment should start with a `#` followed by a single space and should be indented to the same level as the code it precedes. Generally, comments should not be unnecessarily verbose or just state the

obvious, as this can be distracting and can actually make your code more difficult to read!

```
for i in range(0, 10):
    # Loop over i ten times and print out the value of i, followed by a
    # new line character
    print(i, '\n')
```

Bad example

The comments are unnecessary or poorly formatted:

```
def random_walker(T):
    # intalize coords
    x = 0
    y = 0

    for i in range(T):# loop T times
        r = random()
        if r < 0.25:
            x += 1  # go right
        elif r < 0.5:
            x -= 1  # go left
        elif r < 0.75:
            y += 1       # go up
        else:
            y -= 1

        # Print the location
        print((x, y))

    # In Python, the ** operator means exponentiation.
    return x ** 2 + y ** 2
```

Good example

```
def random_walker(T):
    x = 0
    y = 0

    for i in range(T):

        # Generate a random number between 0 and 1.
        # Then, go right, left, up or down if the number
        # is in the interval [0,0.25), [0.25,0.5),
        # [0.5,0.75) or [0.75,1) respectively.

        r = random()
        if r < 0.25:
            x += 1      # Go right
        elif r < 0.5:
            x -= 1      # Go left
        elif r < 0.75:
            y += 1      # Go up
        else:
            y -= 1      # Go down

        print((x, y))

    return x**2 + y**2
```

Whitespaces:

Separate items of a list or tuple by one single space after the comma:

```
# Recommended
my_list = [1, 2, 3]

# Not recommended
my_list = [ 1, 2, 3, ]

# Not recommended
my_list = [1,2,3]
```

Surround the following by single whitespaces on each side:

- Assignment operators, e.g. `=`, `+=`, `-=`, etc.
- Comparison operators, e.g. `<`, `<=`, `!=`, etc.

If there was more than one, surround the lowest priority operator with whitespaces.

```
# Recommended
y = x**2 + 5
z = (x+y) * (x-y)

# Not Recommended
y = x ** 2 + 5
z = (x + y) * (x - y)
```

Don't use whitespaces when assigning argument values:

```
# Recommended
def my_func(default_n=100):
    pass

# Not Recomended
def my_func(default_n = 100):
    pass
```

Treat colons `:` as the operator with the lowest priority **when slicing lists or arrays**:

```
# Recommended
my_list[1:5]
my_list[:5]
my_list[1:]
my_list[::2]
my_list[1:5:2]

# Not recommended
my_list[1 : 5]
my_list[ : 5]
my_list[1: ]
my_list[ ::2]
my_list[1:5 :2]
```

Ooof that was a lot to cover! Thankfully, using **linters** & **formatters** can help you adhere to uniform styling!

(And for a detailed coverage of PEP 8 with various examples, see this)

Linters

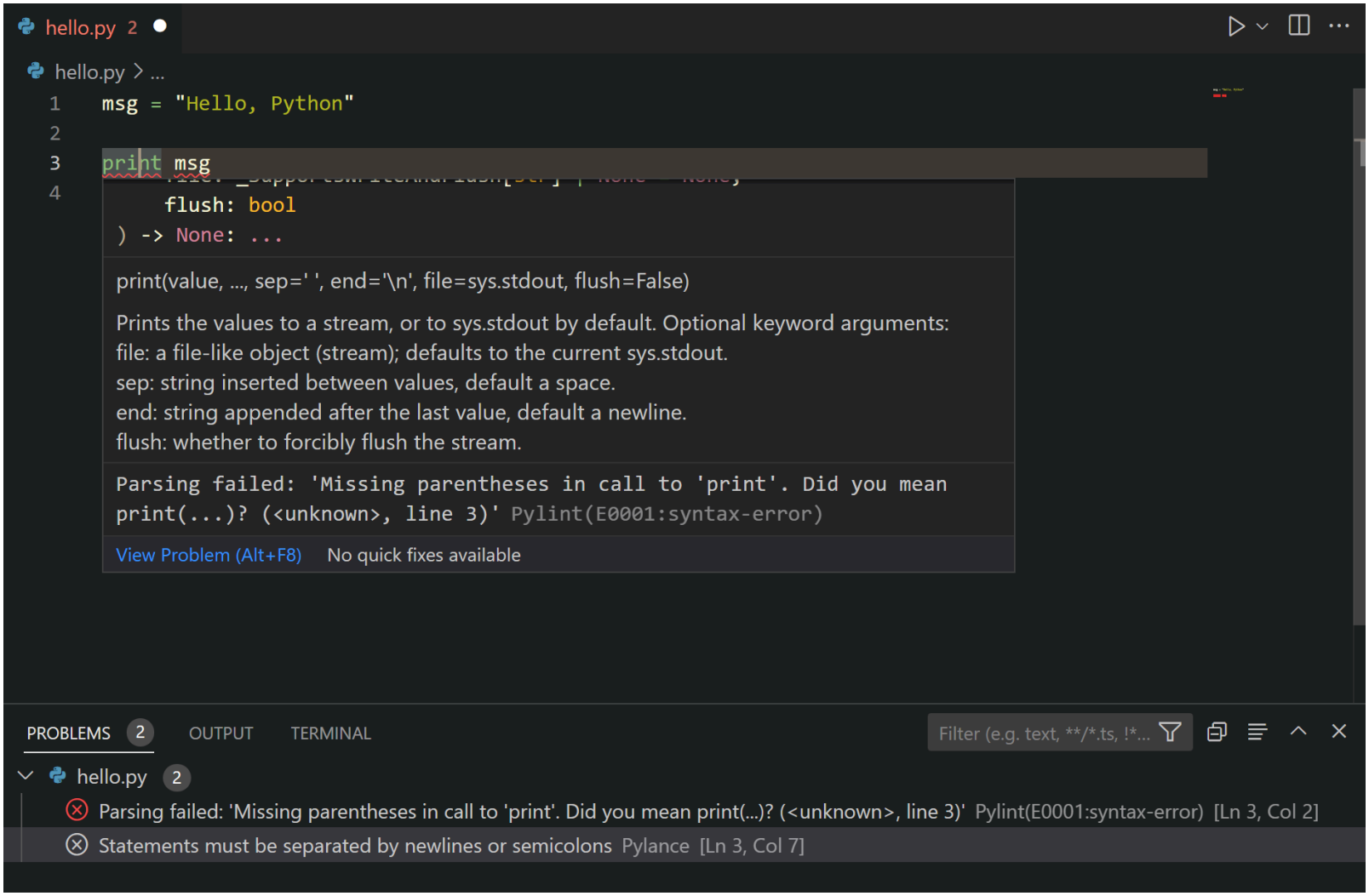
Linting highlights programmatic and stylistic problems in your Python source code. Think of it like "spell check" in word processing software. Common linters for python code include: **pycodestyle**, **pylint**, **pyflakes**, **flake8**, etc.

Some linters only check for programming errors (e.g. pyflakes), whereas some others only verify styling (e.g. pycodestyle), and other ones doing both (e.g. pylint, flake8). For descriptions of each of these linters, see this

Let's enable **Flake8 extension on vscode**

- Open VSCode
- Under extension, search for "Flake8"

- Install and enable the extension



Let's open the file `code/bad_style.py` and see what problems `flake8` highlights.

We can also use `flake8` from the command line to check files

```
flake8 <PATH_TO_FILE>
```

Let's try this in the terminal. You may first need to run `conda install flake8`.

Formatters

Formatting: restructures how code appears by applying specific rules for line spacing, indents, line length, etc. Common python auto formatters include **autopep8**, **black**, **yapf**, etc. Many of these auto formatters are available as an extension in vscode.


https://i.stack.imgur.com/oyG5K.png

`black` can also be used from the command line to format your files:

```
black <PATH_TO_FILE> --check
```

The `--check` argument just checks if your code conforms to black style but doesn't reformat it in place, if you want your file reformatted, remove the argument.

Before formatting:

```
x = {  'a':37,'b':42,
'c':927}
very_long_variable_name = {'field': 1,
                           'is_debug': True}
condition=True

if very_long_variable_name is not None and very_long_variable_name["field"] > 0 or very_long_variable_n
 z = 'hello '+'world'
else:
 world = 'world'
 a = 'hello {}'.format(world)
 f = rf'hello {world}'
if (condition): y = 'hello ''world' #FIXME: https://github.com/python/black/issues/26
class Foo  (     object  ):
  def f    (self   ):
    return       37*-2
  def g(self, x,y=42):
      return y
# fmt: off
custom_formatting = [
    0,  1,  2,
    3,  4,  5
]
# fmt: on
regular_formatting = [
    0,  1,  2,
    3,  4,  5
]
```

After formatting:

```
x = {"a": 37, "b": 42, "c": 927}
very_long_variable_name = {"field": 1, "is_debug": True}
condition = True

if (
    very_long_variable_name is not None
    and very_long_variable_name["field"] > 0
    or very_long_variable_name["is_debug"]
):
    z = "hello " + "world"
else:
    world = "world"
    a = "hello {}".format(world)
    f = rf"hello {world}"
if condition:
    y = "hello " "world"  # FIXME: https://github.com/python/black/issues/26


class Foo(object):
    def f(self):
        return 37 * -2

    def g(self, x, y=42):
        return y


# fmt: off
custom_formatting = [
    0,  1,  2,
    3,  4,  5
]
# fmt: on
regular_formatting = [0, 1, 2, 3, 4, 5]
```

Guidelines that cannot be checked automatically

- Variable names should use underscores (PEP 8), but also need to make sense.

    - e.g. `spin_times` is a reasonable variable name
    - `my_list_of_thingies` adheres to PEP 8 but is NOT a reasonable variable name
    - same for `lst`—fine for explaining a concept, but not as part of a script that will be reused
- DRY
- Magic numbers
- Comments

(OPTIONAL) 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 (recall last lecture we discussed how functions are objects in Python and can be passed around into other functions!)
- 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 execute 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 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 lecture


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 0.786 sec
```

```

```

- Python's built-in decorators like `classmethod` and `staticmethod` are coded in C so I'm not showing them here
- You may not often create your own decorators, but many built-in can come in handy!

(OPTIONAL) Getters/setters/deleters

- There's one more import topic to talk about with Python classes and that is getters/setters/deleters
- (You might be familiar with these terms coming from other OOP languages)
- The necessity for these actions is best illustrated by example

- Here's a stripped down version of the `MdsMember` class from earlier

```
class MdsMember:

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.initials = (first[0] + last[0]).upper()

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

```
mds_1 = MdsMember('Quan', 'Nguyen')
print(mds_1.first)
print(mds_1.last)
print(mds_1.initials)
print(mds_1.full_name())
```

```
Quan
Nguyen
QN
Quan Nguyen
```

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

```
mds_1.first = 'Juan'
print(mds_1.first)
print(mds_1.last)
print(mds_1.initials)
print(mds_1.full_name())
```

```
Juan
Nguyen
QN
Juan Nguyen
```

- Uh oh... the initials 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 `initials()` 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 reference to the `initials` attribute to a call to the `initials()` 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 `initials` like a method, but keep it as an attribute using the `@property` decorator

```
class MdsMember:

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

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

    @property
    def initials(self):
        return (self.first[0] + self.last[0]).upper()
```

```
mds_1 = MdsMember('Quan', 'Nguyen')
mds_1.first = 'Juan'
print(mds_1.first)
print(mds_1.last)
print(mds_1.initials)
print(mds_1.full_name())
```

```
Juan
Nguyen
JN
Juan Nguyen
```

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

```
class MdsMember:

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

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

    @property
    def initials(self):
        return (self.first[0] + self.last[0]).upper()
```

```
mds_1 = MdsMember('Juan', 'Nguyen')
mds_1.full_name
```

```
'Juan Nguyen'
```

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

```
mds_1.full_name = 'Quan Nguyen'
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[94], line 1
----> 1 mds_1.full_name = 'Quan Nguyen'

AttributeError: property 'full_name' of 'MdsMember' object has no setter
```

- 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 MdsMember:

    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 initials(self):
        return (self.first[0] + self.last[0]).upper()
```

```
mds_1 = MdsMember('Quan', 'Nguyen')
mds_1.full_name = 'Juan Nguyen'
print(mds_1.first)
print(mds_1.last)
print(mds_1.initials)
print(mds_1.full_name)
```

```
Juan
Nguyen
JN
Juan Nguyen
```

- Almost there! We've talked about getting information and setting information, but what about deleting 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 MdsMember:

    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 initials(self):
        return (self.first[0] + self.last[0]).upper()
```

```
mds_1 = MdsMember('Quan', 'Nguyen')
delattr(mds_1, "full_name")
print(mds_1.first)
print(mds_1.last)
```

```
Name deleted!
None
None
```