Introduction

Welcome to Intermediate Topics in Python! The focus of this lesson will be writing cleaner and more maintainable Python code.

Agenda: Functions Classes

The DRY Principle

Don't **R**epeat **Y**ourself

If you need to repeat a process, make sure you're not reusing code you've already written. A programmer's goal is to be as lazy as possible.

Using loops, functions, and classes are all ways of condensing your code.

Functions

Why we need functions

Say we are given a string and we need to generate its character map.

For instance, if we are given the string "Hello", we want to generate the following dictionary:

```
{
    "H": 1,
    "e": 1,
    "l": 2,
    "o":, 1
}
```

If we now want to do this for several strings, a naive solution without functions is as follows:

```
hello_dict = {}
hello = "Hello"

for c in hello:
    if c in hello_dict:
        hello_dict[c] += 1
    else:
        hello_dict[c] = 1

foo_dict = {}
foo = "foo"

for c in foo:
    if c in foo_dict:
        foo_dict[c] += 1

    else:
        foo_dict[c] = 1
```

While this may not seem too bad for two strings, imagine doing this for thousands of strings. This is where functions come in.

Basic function syntax

The syntax for defining a function is as follows:

```
def foo(num):
```

```
        print("bar" * num)
```

`def` specifies that we are defining a function.

`foo` is the name of the function.

`num` is a parameter, which is just some data that gets passed in.

Since there is no `return` statement, this function returns nothing.

Given this function, if we want to see "barbarbarbar" in our console, we would do

```
    foo(4)
```

When we pass 4 as a parameter, `num` becomes equal to 4 for the remainder of the function. Once we get out of the function, `num` will no longer exist.

What if we want to take `x`, perform `foo(x)`, and then perform `bar()` on this new value? We'll have to use a `return` statement for this.

```
    def foo(num):
        return "bar" * num

    def bar(string):
        return string[::-1]

    print(foo(2)) # prints barbar
    print(bar("barbar")) # prints rabrab

    print(bar(foo(2)) # also prints rabrab
```

Python is a duck-typed language, so there's no need to specify a function's return type or the types of its parameters.

In [6]:

```
def foo(num):
    print("0x", "fox" * num)
foo(4)
```

```
0x foxfoxfoxfox
```

```
    hello_dict = {}
    hello = "Hello"

    for c in hello:
        if c in hello_dict:
            hello_dict[c] += 1
        else:
            hello_dict[c] = 1

    foo_dict = {}
    foo = "foo"

    for c in foo:
        if c in foo_dict:
            foo_dict[c] += 1

        else:
            foo_dict[c] = 1
```

Condensing the provided code with functions

In [7]:

```
def get_char_map(string):
    string_dict = {}
```

```
    for c in string:
        if c in string_dict:
            string_dict[c] += 1
        else:
            string_dict[c] = 1

    return string_dict

print(get_char_map("Hello"))
```

```
{'H': 1, 'e': 1, 'l': 2, 'o': 1}
```

Intermediate function syntax

Optional arguments

Python supports optional arguments for functions. The way these work is relatively straightforward - you can set some default value for a parameter `x`, but if a user specifies a value for `x`, then that value will be used.

Example:

In [13]:

```
def foo(**kwargs):
#     print(**kwargs)
    print("bar" * kwargs["num"])

# foo()
foo(num=2)
```

```
barbar
```

Type hints

Type hints are Python's way of making duck typing more readable. With type hints, we can specify the expected value of our function's parameters and return value.

Take the code we wrote for the character map, for instance.

```
    def char_map(string):
        char_map = {}

        for c in string:
            if c in char_map:
                char_map[c] += 1
            else:
                char_map[c] = 1

        return char_map
```

To specify to the user that the input type will be a `str`, and the return value will be a `dict`. To do so, we will change the function declaration to

```
    def char_map(string: str) -> dict:
```

Keep in mind that type hints have no effect on the execution of the program. A user can still pass an integer or a list as a parameter of the function.

Classes

Introduction to classes

Introduction to classes

Classes are a way to make your code focus on data, rather than actions. Classes group your data with possible actions it may perform.

Say we want to create a model for every student in Binghamton University. Every student has a name, a B-Number, and a major. We should also keep track of the student's position, and give them a `go_to_class` method so they don't fail all their classes.

We can model this information with Python as follows:

```
class Student:
    def __init__(self, name: str, b_number: str, major: str, position: str) -> None:
        self.name = name
        self.b_number = b_number
        self.major = major
        self.position = position

    def go_to_class(self, class_location: str) -> None:
        self.position = class_location


tom = Student("Tom", "B1001001", "Computer Science", "EB G7")
tom.go_to_class("Fine Arts 212")
```

In [15]:

```
class Student:
    def __init__(self, name: str, b_number: str, major: str, position: str) -> None:
        self.name = name
        self.b_number = b_number
        self.major = major
        self.position = position
        self.go_to_class(position)

    def go_to_class(self, class_location: str) -> None:
        self.position = class_location


tom = Student("Tom", "B1001001", "Computer Science", "EB G7")
print(tom.position)
tom.go_to_class("Fine Arts 212")
print(tom.position)
```

```
EB G7
Fine Arts 212
```

What's happening here?

`class Student` declares a class called Student.

`__init__` is the constructor of the object. It takes 4 parameters, the student's:

1. Full name
2. B-Number
3. Major
4. Position

and sets the passed values to be its instance variables.

We also declare a method called `go_to_class`. A method is just a function that belongs to a particular class.

`go_to_class` changes the `self.position` instance variable to whatever the user passes. Hopefully it's the location of their classroom, not The Rat.

We then create a `Student` object called `tom`, and pass Tom's information into the parameters.

Lastly, we disrupt Tom's programming marathon in EB G7 and force him to go to class in the Fine Arts building.

Let's try an example

Create a class `Dog` that models a specific dog. It should hold a dog's name, breed, and year of birth. It should also have a method `bark()`, which prints "woof" to the console.

In [17]:

```
class Dog:
    def __init__(self, name: str, breed: str, year_of_birth: int) -> None:
        self.name = name
        self.breed = breed
        self.year_of_birth = year_of_birth

    def bark(self):
        print("woof")
```