Defining Classes

WE'LL COVER THE FOLLOWING

- The `__init__()` Method

Python is fully object-oriented: you can define your own classes, inherit from your own or built-in classes, and instantiate the classes you've defined.

Defining a class in Python is simple. As with functions, there is no separate interface definition. Just define the class and start coding. A Python class starts with the reserved word `class`, followed by the class name. Technically, that's all that's required, since a class doesn't need to inherit from any other class.

```
class PapayaWhip:  #①
    pass           #②
```

① The name of this class is `PapayaWhip`, and it doesn't inherit from any other class. Class names are usually capitalized, `EachWordLikeThis`, but this is only a convention, not a requirement.

② You probably guessed this, but everything in a class is indented, just like the code within a function, `if` statement, `for` loop, or any other block of code. The first line not indented is outside the class.

This `PapayaWhip` class doesn't define any methods or attributes, but syntactically, there needs to be something in the definition, thus the `pass` statement. This is a Python reserved word that just means "move along, nothing to see here". It's a statement that does nothing, and it's a good placeholder when you're stubbing out functions or classes.

> *The **pass** statement in Python is like a empty set of curly braces (**{}**) in Java or C.*

Many classes are inherited from other classes, but this one is not. Many classes define methods, but this one does not. There is nothing that a Python class absolutely must have, other than a name. In particular, C++ programmers may find it odd that Python classes don't have explicit constructors and destructors. Although it's not required, Python classes can have something similar to a constructor: the `__init__()` method.

The `__init__()` Method

This example shows the initialization of the `Fib` class using the `__init__` method.

```
class Fib:
    '''iterator that yields numbers in the Fibonacci sequence'''  #①

    def __init__(self, max):                                      #②
        self.max = max
```

① Classes can (and should) have `docstrings` too, just like modules and functions.

② The `__init__()` method is called immediately after an instance of the class is created. It would be tempting — but technically incorrect — to call this the "constructor" of the class. It's tempting, because it looks like a C++ constructor (by convention, the `__init__()` method is the first method defined for the class), acts like one (it's the first piece of code executed in a newly created instance of the class), and even sounds like one. Incorrect, because the object has already been constructed by the time the `__init__()` method is called, and you already have a valid reference to the new instance of the class.

The first argument of every class method, including the `__init__()` method, is always a reference to the current instance of the class. By convention, this argument is named `self`. This argument fills the role of the reserved word `this` in C++ or Java, but `self` is not a reserved word in Python, merely a naming convention. Nonetheless, please don't call it anything but `self`; this is a very strong convention.

In all class methods, `self` refers to the instance whose method was called. But

In all class methods, `self` refers to the instance whose method was called. But in the specific case of the `__init__()` method, the instance whose method was called is also the newly created object. Although you need to specify `self` explicitly when defining the method, you do not specify it when calling the method; Python will add it for you automatically.