Recap: how Python works

When we run the Python interpreter on a file that contains a valid program, Python starts at the first line of the program and starts executing. As the program runs, Python introduces and modifies bindings in the appropriate namespace, according to the rules described below.

Python creates namespaces at the start of a program and at function calls

When Python starts up, it creates a built-in namespace and populates it with bindings for all the built-in functions.

Next, Python creates an empty, global namespace.

Any time Python sees a function call, it creates a new local namespace for that function and pushes the namespace onto a stack of local namespaces. When the function returns, Python pops the corresponding local namespace off the stack (and destroys it).

There is only one, currently active namespace. When Python is executing the body of a function, the currently active namespace is the one on the top of the local-namespace stack. When Python is not executing the body of a function, the currently active namespace is the global namespace.

An assignment creates a binding in the currently active namespace

When Python sees a statement like this:

\[ x = 3 \]

Python binds the name \( x \) to the value 3 in the currently active namespace.

This statement is an example of an assignment. All assignments have the form

\[ name = expression \]

where \( name \) is a valid Python variable name and \( expression \) is a valid Python expression.

Name resolution: local, global, builtin, error

When a running program refers to a variable's name, Python will try to look up the value for that name. (Instead of saying “Python looks up the value for a name”, we often say “Python resolves the name”.) Python always runs the same algorithm to resolve a name:

1. **Local**: If there is an active local namespace (i.e., if the local-namespace stack is not empty), look for a binding in the namespace at the top of the stack. If a binding exists for the name, use the corresponding value.

2. **Global**: If there is no active local namespace or if resolution fails for the local namespace, then look for a binding in the global namespace. If a binding exists for the name, use the corresponding value.

3. **Builtin**: If resolution fails for the global namespace, then look for a binding in the builtin namespace. If a binding exists for the name, use the corresponding value.

4. **Error**: If resolution fails for the builtin namespace, throw a NameError.

---

1The information in this sketch is a necessary simplification of the complete rules of Python. However, these rules cover 99.9% of well-written Python code that you’re likely to see. Furthermore, the complete rules are just extensions to these rules, and the extensions are fairly easy to understand once you know the things in this sketch.
Modules: they’re just more namespaces!

Resolving a module’s name

When Python sees a statement like this:

```python
import numbers
```

Python looks for a file called `numbers.py` using the following procedure:

1. Look for the file in the current directory. If found, proceed with the `import`.
2. If there is no such file in the current directory, look in the module search path. If found, proceed with the `import`.
3. If the file isn’t in the module search path, throw an `ImportError`.

An import statement binds a name to a namespace (!!)

When Python sees the line:

```python
import numbers
```

it performs the following steps:

1. Create a new namespace (which we’ll refer to as `N`).
2. Run `numbers.py` as if it were a program, using `N` as the global (and currently active) namespace.
3. Bind the name `numbers` to `N` in the originally active namespace.

Other forms of import also affect bindings

Python provides several forms of the `import` statement, each of which gives you a certain amount of control over how names from the module are bound.

When Python sees the line:

```python
from numbers import double
```

it does steps 1 and 2 from above, then it does the following:

3. Copy the binding for `double` to the originally active namespace.

When Python sees the line:

```python
from numbers import double, triple
```

it does steps 1 and 2 from above, then it does the following:

3. Copy the bindings for `double` and `triple` to the originally active namespace.

When Python sees the line:

```python
import numbers as num
```

it does steps 1 and 2 from above, then it does the following:

3. Bind the name `num` to `N` in the originally active namespace.

---

2 The module search path is a list of directories that includes the standard-library directories. We almost certainly won’t need to configure this list in CS 42; but if you’re interested, there’s more information here: bit.ly/2e7LyBe.
Object-oriented programming (OOP) in Python

Terminology

The terms below are the ones that the Python documentation uses most often, when referring to object-oriented concepts. These terms are themselves defined using the language-agnostic definitions from today’s other handout. Note that languages other than Python—as well as some Python programmers and resources—might use different terms than the ones below.

- **object**
  any value in Python

- **class**
  describes the interface and implementation of an object

- **instance**
  an object that has been created from a class’s description
  
  *We usually use the word “instance” to mean a user-defined object rather than a built-in one such as a number, list, or dictionary.*

- **type (of an object)**
  the class used to create that object

- **method**
  describes a behavior of an instance

- **instance variable**
  a field of an instance

- **attribute**
  either a method, property, or field of an instance

- **class variable**
  an instance variable defined in the namespace of a class (rather than in the namespace of an instance of that class). All instances of a class share the same set of class variables.

- **self**
  the name of an object’s reference to itself

- **inheritance**
  if class B inherits from class A, then class B has access to the interface and the implementation of class A, through self

Gotchas

If you’re used to OOP in other languages these things in Python might seem weird at first.

- **All methods (including the constructor) must explicitly declare self as the first parameter.**
  This rule makes it easy to distinguish attributes from local variables, in a method body. (If you’re used to Java, forgetting to declare self as the first parameter of all your methods might be your most common mistake, when you start to write object-oriented Python programs.)

- **Even though methods explicitly declare a self parameter, method calls don’t take an explicit argument for the instance.**
  Instead, Python makes sure that self is bound to the instance. (If you’re used to Java, this means that—despite the weirdness of method declarations—Python method calls work just the same as Java ones.)

- **In a method body, we must use self to access attributes of an instance.**

- **By convention, we use an underscore at the beginning of the name of any attribute that corresponds to an implementation detail (e.g., fields or private helper methods), like so: self._data.**
  (There are no public and private declarations in Python.)

- **The constructor is called __init__. The __str__ method is like Java’s toString. Both these methods are examples of “special methods”, and they’re Python’s way of describing operations (such as initialization or converting to a string) that many instances might want to support.**

---

3 Technically, the first parameter doesn’t need to be named self, but the convention is to do so. In fact, it’s so much of a convention in Python programming that you should treat it as rule of the language.
Classes are (you guessed it!) just namespaces

When Python sees a statement that starts like this:

class Course:
    ...class body...

it performs the following steps:

1. Create a new namespace (which we'll refer to as $N$).
2. Run the class body as if it were a function body, using $N$ as the local (and currently active) namespace.
3. Bind the name Course to $N$ in the originally active namespace.

Instances are (yep!) just namespaces

When Python sees a statement like this:

cs42 = Course(42, 'Principles and Practices of Computer Science')

it performs the following steps:

1. Create a new namespace (which we'll refer to as $N$).
2. Call Course.__init__ as if it were a function, binding self to $N$ in the body of the local namespace for __init__, along with the bindings for the other parameters.
3. Bind the name cs42 to $N$ in the originally active namespace.

Assigning to an attribute creates a binding in the instance’s namespace

When Python sees a statement like this:

cs42.number = 1000

Python binds number to the value 1000 in cs42’s namespace.

This statement is an example of an object modification. Most object modifications have the form

name.attribute = expression

where name and attribute are a valid Python variable names, name is bound to an instance, and expression is a valid Python expression. Note that this rule means that instances are mutable.

Attribute resolution: instance, class, superclass(es)

When a running program refers to an instance’s attribute, Python will try to look up the value for that attribute’s name. Python always runs the same algorithm to resolve an attribute’s name:

1. **Instance**: Look for a binding in the namespace of the instance. If a binding exists for the attribute’s name, use the corresponding value.
2. **Class**: If resolution fails for the instance’s namespace, then look for a binding in the namespace of the instance’s class. If a binding exists for the attribute’s name, use the corresponding value.
3. **Superclass(es)**: If resolution fails for the instance’s class, then look for a binding in the namespace(s) of the instance’s superclass(es). If a binding exists for the attribute’s name, use the corresponding value.
4. **Error**: If resolution fails for the superclass(es), throw an AttributeError.

Next time: no class (study for or take the midterm)!