CIS 343 - Structure of Programming Languages

Nathan Bowman

Slides by (with slight modifications): Ira Woodring

Language Spotlight - Python A Strong, Dynamically Typed Language

Python

Python was created by Guido van Rossum, a Dutch programmer. He retains the title "Benevolent Dictator for Life" (BDFL) of the Python language.

Python now maintains just one version - Python 3. Until January 2020, both Python 2.7 and Python 3 were actively maintained, but all support, including bugfixes, for Python 2 ended at that point. You may see Python 2 code, as it was very popular, but all of your code should use Python 3.

The name gives a nod to *Monty Python's Flying Circus*.

Python

The language is widely used - usually in the top 10 on the Tiobe index (#3 as of the time of this writing). It is used by companies such as

- Youtube
- Industrial Light & Magic (Lucasfilm!)
- Google,
- Firaxis Games (Civilization Series)
- MANY others.

Python

Python allows programmers to program in multiple paradigms; imperative, procedural, and object-oriented paradigms are supported. The language also provides some functional features.

Types in Python are dynamic, meaning variables are not bound before runtime. This makes writability much easier for programming in Python, but affects readability and reliability (due to a lack of compile time type checking).

Python is strongly typed (though people are often confused about this because we can change the type ourselves so easily). The reason it is strongly typed is that variables are not automatically coerced into other types. Consider:

```
a = "42"
b = 1 + a

c = 42
print("The answer is " + c)
```

van Rossum continues to oversee development of Python due to his status as BDFL, but the language is influenced by these general goals:

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated
- Readability counts

```
import this
```

More generally, Guido believes "there should be one—and preferably only one—obvious way to do it" when it comes to programming.

Python

Many of those design decisions can be seen by the syntax that Python chooses to use. Whitespace is important; therefore indentation is required. English words are often used in place of symbols (i.e. `not` instead of `!`).

Additionally, due to these design decisions Python is more orthogonal than many other languages like C.

Python

Python is an interpreted language, meaning programs do not need to be compiled before run. This provides the added benefit of being able to interactively use Python (demo).

Python

Python has several built in variable types: Numeric, Iterator, Sequence, Text Sequence, Binary Sequence, Set, Mapping, Class, and a few other types.

Python

Variables are given a type implicitly and dynamically; this means that variables are not declared with a type modifier as in some languages, and that memory for the variable is not reserved until run-time.

Python

Numeric types may be

int - Integers (1, 2, 3, 42, 9192, etc.)

float - Floating Point (3.14, 42.424242)

complex - Complex (6+8j) (they have a real part i and an imaginary part j). We don't need to put the i before the real part.

Python

Many functions are provided by Python for use on Numeric types. Some include

```
abs(x)
int(x)
float(x)
pow(x)
-x
```

etc. (note some of these are not defined for complex numbers though). We do not need to import any additional libraries to use these functions.

Python

Iterator types allow programmers to iterate over a collection. We will talk about iterators later in the semester. An iterator basically points to a current object and provides methods to point to the next (and sometimes previous) object.

In Python any object that is iterable must have the two methods

```
__iter__()
__next__()
```

must be defined.

Python

Sequence types in Python come in three varieties, **list**, **tuple**, and **range**. Each of these are iterable.

Python

The list in Python is very similar to an array (reinforced by the fact that lookups on the list take O(1) time! Documentation). They are used much like arrays, but have enhanced functionality over many of their peer languages' more basic arrays.

Python

Some built-in functions that Python provides for lists are

- `list.append(x)` - add to the end of the list
- `list.insert(i,x)` - add the element to the list at position i. This will move other elements in the list.
- `list.clear()` - Removes all items from the list.
- `list.sort`, `list.count`, and other built-ins are provided as well (demo).

For those of you interested, the sort method makes use of Timsort.

It is a derivation of merge and insertion sort.

It is also interesting to note that you can sort a list by calling the `sort()` method on the object, or you can merely return a sorted representation with the `sorted(list)` function.

Python

The built-in functionality of lists allows them to easily be used as stacks and queues.

Remember, a stack is a LIFO data structure, while a queue is a FIFO.

Python

Stack operations:

```
stack = []
stack.append[42]
stack.append[492727]
stack.pop()
```

Python

Queue Operations:

```
queue = []
queue.append(42)
queue.append(492727)
q.pop(0)
```

Or, with an additional import statement:

```
from collections import deque
queue = deque([42, 492727])
queue.append(1701)
queue.popleft()
```

Python

Python provides an interesting way to create lists as well, called **List Comprehensions**. We can create these by applying a function to a list.

If we were normally going to create a list in a for loop like this:

```
nums = []
for i in range(100):
  nums.append(i*2)
```

We could instead use this (more concise) code:

```
nums = [x*2 for x in range(100)]
```

Python

Additionally, Python would also let us take a functional programming approach, and map a function:

```
nums = list(map(lambda x: x*2, range(10)))
```

Functional programming relies upon mapping functions to an input instead of the for-each syntax we are more accustomed to in imperative programming.

Python

Tuples are another sequence type in Python, but whereas list elements can change, tuple elements are immutable.

We create tuples with parentheses, separating each element in the tuple by a comma:

```
''' Technically the ( and ) are optional here... '''
tup = (42, 492727, 1701)
tup = ([42, 492727], 1702)
tup = (["Meaning", 42], 492727)
```

Python

We access data inside a tuple with our indexing operators, "[", and "]".

```
>>> tup
(['Meaning', 42], 492727)
>>> tup[0]
['Meaning', 42]
>>>
```

Python

Remember though! Tuples are immutable. This means we cannot change them once they have been created:

```
>>> tup[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

Python

Python provides support for Sets as well (these are not sequence types!). Sets are unordered collections that have no duplicate elements. We define sets with curly braces:

```
>>> students = {'henrietta', 'isaac', 'ira', 'castiel'}
>>> 'ira' in students
True
>>> 'mia' in students
False
```

Python

Dictionaries (mapping types) are also supported:

```
>>> students = {'henrietta':1928, 'isaac': 1958, 'ira':9, 'castie
>>> students[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
>>> students['ira']
9
>>> students.keys()
['isaac', 'henrietta', 'castiel', 'ira']
```

Python

While looping through dictionaries, it is often useful to have access to both the key and the value. Python makes this easy:

```
>>> for s,n in students.items():
...     print(s,n)
...
('isaac', 1958)
('henrietta', 1928)
('castiel', 3)
('ira', 9)
```

Python

The *Text Sequence* type is how Python provides functionality for strings. These sequences are `str` objects comprised of Unicode values.

str objects are immutable!

```
>>> name="ira"
>>> name
'ira'
>>> name[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

Python

They can be created with single, double, or triple quotes:

```
name = 'ira'
name = "ira"
name = '''ira'''
name = """ira"""
```

Triple quotes allows us to span multiple lines; all text (including whitespace) will be included in the `str` object.

Python

Thought they are immutable, but Python provides easy operators to construct new objects:

```
name = "ira"
l_name = "woodring"
n = name + l_name
```

Python

Be careful though! Some of the provided functions may not do what you think they would do. Take str.join() for instance. We might expect it to combine multiple strings - and it does. At least, in a way:

```
>>> name
'ira'
>>> l_name="woodring"
>>> str.join(name, l_name)
'wiraoiraoiradirariraiiranirag'
```

Python

Many functions for working on text sequences are provided by Python. Such as

```
str.capitalize()
str.count()
str.expandtabs()
str.find()
str.isalpha()
str.isdecimal()
str.islower()
str.replace()
```

And many more.

Python

Binary Sequence Types are supplied for dealing with binary data. These types are

- **bytes** - immutable sequences of single bytes.
- **bytearray** - mutable sequences of bytes.
- **memoryview** - allows us to see the internal memory of another object. This is used with the Python C-API, which allows us to write C or C++ code that we can run with Python.

Python

We will focus mostly on the bytearrays, since they are mutable:

```
>>> b
bytearray(b'\xa1\xb2\xc3\xd4\xe5\xf6\x00\xff')
>>> b[0]
161
>>> b[0] = 163
>>> b
bytearray(b'\xa3\xb2\xc3\xd4\xe5\xf6\x00\xff')
>>>
```

A plethora of functions for working on these bytearrays exist, but we will not discuss them here.

Python

Functions in Python are created with the `def` keyword.

As with all of Python's scope blocks, we must end the declaration with a ":" and indent subsequent lines:

```
def my_func():
  print "Hello World!"
```

Python

Python allows multiple methods of passing parameters.

This includes the "normal" ways we are used to in Java, but also keyword parameters and variable number of parameters.

Python

In our projects for this class we are going to focus on Object Oriented Python. The syntactic unit for defining an object in Python is the class.

```
class Starship:
  """A class to store Starship information."""
  num_ships = 0
  def __init__(self):
    """Class initializer code."""
    self.number_crew = 0
    self.shield_level = 100
    self.name = ""
    self.crew = []
    Starship.num_ships = Starship.num_ships + 1
  def set_name(self, name):
    """Sets the ship's name."""
    self.name = name
```

Python

In the previous example `num_ships` is a class (static) variable. It is therefore shared amongst all instances of the class.

Python

Unlike in Java and C++, in Python we are able to declare our instance variables at any point in the runtime. A good place to put our instance variables is in our `__init__` function; this enhances readability and reliability, as the general consensus in Python programming is to both put our instance variables here and to initialize them to a default state.

Python

You may have noticed the strings

```
"""A class to store Starship information."""
"""Class initializer code."""
"""Sets the ship's name."""
```

These are __doc__ strings. Think of them like Javadocs. You can examine the docstring for any class or module that provides them as such:

```
Starship.__doc__
Starship.set_name.__doc
```

etc.

Python

You may be wondering how to have an overloaded constructor in Python; unlike many other languages this isn't as straightforward. Overloaded constructors are not directly possible in Python, but there are multiple ways to have equivalent functionality.

We aren't going to go over those here, but there are many articles online concerning the issue.

Python

It is important to reinforce that any object in Python can have a member added at any time:

```
>>> class Blah:
...     pass
...
>>> b = Blah()
>>>
>>> def say_name(name):
...     print name
...
>>> b.say_name = say_name
>>> b.say_name('ira')
ira
```

Python

In Python programmers are simply expected not to write to another class's variables. It is a cultural sort of thing; there is nothing preventing you from doing so.

This means that Python doesn't *really* have private variables. This fits with Python's philosophy of their being a "right way" to program. It is then up to the programmer to do the right thing.

Python

You can *sort of* have private variables in Python by using Python's variable mangling abilities. Variable mangling occurs when we prefix a variable we wish to be private with the ___ characters. This causes Python to internally rename the variable (you can read up on how this happens but we will not discuss it in class).

This "mangling" of variables is a useful way to make data harder to access, but isn't perfect. It complicates things. Often, Python programmers use a single underscore to prefix a variable. This tells other Python programmers it should be considered private.

In general you should prefix "private" variables with one of those conventions.

```
class Blah:
    def __init__(self):
        self.__stuff = 42

>>>b = Blah()
>>>b.__stuff
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Blah' object has no attribute '__stuff'

>>> b._Blah__stuff
42
```

We can still modify this. But we shouldn't.

Python

Python does support inheritance. We inherit from a class as such:

```
>>> class Mammal(object):
        def __init__(self):
            self.type = "Mammal"

>>> class Dolphin(Mammal):
        def __init__(self):
            super(Dolphin, self).__init__()

>>> Dolphin()
<__main__.Dolphin object at 0x10e212bd0>
>>> d = Dolphin()
>>> d.type
'Mammal'
```

Python

Python provides facilities for bringing other symbols into the namespace with the `import` statement. These can take multiple forms:

```
# Import this library into the current namespace
import sys
# Import this library into a namespace called s
import sys as s

# Import this code from a namespace
from sys import argv
# Import this code from a namespace and bind it to a new name
from sys import argv as a
```

Python

Creating your own libraries in Python is quite simple. You merely create a directory with the name of your package, and include a file called `__init__.py`. This file is often empty (but doesn't have to be).

```
mkdir my_lib
cd my_lib
touch __init__.py

$ python
Python 2.7.10 (default, Feb  7 2017, 00:08:15)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)] on darwi
Type "help", "copyright", "credits" or "license" for more informa
>>> import my_lib
>>>
```

Python

Currently our package is empty. We could change that by adding the file `stuff.py` into the directory `my_lib`.

```
def sayHi(num):
  for i in range(num):
    print "Hi."
```

Python

We can now use this code by importing it as such:

```
>>> from my_lib import stuff
>>> stuff.sayHi(5)
Hi.
Hi.
Hi.
Hi.
Hi.
```

The **all** variable in the **init**.py file controls which submodules are loaded when we type

```
from my_lib import *
```

Python

An understanding of **the Observer Pattern** is necessary for completing the project. Note that this **does not** pertain just to Python, this is simply a programming paradigm and can be used with many languages. It is included in the Python section for lack of a better option.

Python

This pattern is implemented by having an object keep a list of other objects that should be notified upon state changes. The other objects are called observers to the object.

Python

To implement this pattern in Python it is useful to create some library code we can reuse later:

An abstract Observer class:

`observer.py`

```
from abc import ABCMeta, abstractmethod

class Observer(object):
        __metaclass__ = ABCMeta

        @abstractmethod
        def update(self):
                pass
```

Note!

This particular interface doesn't pass anything to the observers. We may want to pass different types of messages though; this class would then need to be extended.

Observable class (with no parameters passed to observers):

```
class Observable(object):

        def __init__(self):
                self.observers = []

        def add_observer(self, observer):
                if not observer in self.observers:
                        self.observers.append(observer)

        def remove_observe(self, observer):
                if observer in self.observers:
                        self.observers.remove(observer)

        def remove_all_observers(self):
                self.observers = []

        def update(self):
                for observer in self.observers:
                        observer.update()
```

And a demo:

```
from observer_pattern.observer import Observer
from observer_pattern.observable import Observable

class House(Observer):
    def update(self):
        print "Monster updated!"

class Monster(Observable):
    pass

if __name__ == "__main__":
    h = House()
    vampire = Monster()
    mummy = Monster()

    vampire.add_observer(h)
    mummy.add_observer(h)

    vampire.update()
    mummy.update()
```

Credit: My code is a modification of code from a blog post on the Observer pattern in Python. The post was formerly found here, but appears to no longer exist.