Python

Jarek Szlichta

http://data.science.uoit.ca/

Python Language

- **Python is a straightforward language**
  - very simple syntax
- **It encourages programmers to program without boilerplate (prepared) code**

Variables and Types

- **Python is completely object oriented, and not "statically typed“**
  - You do not need to declare variables before using them, or declare their type
  - Every variable in Python is an object

Numbers

- Python supports two types of numbers - integers and floating point numbers
  - To define an integer, use the following syntax:

```
myint = 7
```

  - To define a floating point number, you may use one of the following notations:

```
myfloat = 7.0
myfloat = float(7)
```

Strings

- **Strings are defined either with a single quote or a double quote.**

```
mystring = 'hello'
mystring = "hello"
```

- **Using double quotes makes it easy to include apostrophes**
  - whereas these would terminate the string if using single quotes

```
mystring = "Don't worry about apostrophes"
```

Lists

- Lists are very similar to arrays
  - They can contain any type of variable,
  - and they can contain as many variables as you wish
- Lists in Python can also be iterated over in a simple manner

Lists

```
# empty list
mylist = []
mylist.append(1)
mylist.append(2)
mylist.append(3)
print(mylist[0]) # prints 1
print(mylist[1]) # prints 2
print(mylist[2]) # prints 3

# prints out 1,2,3
for x in mylist:
    print x
```

Exceptions

- **Accessing an index which does not exist generates an exception (an error)**

```
mylist = [1,2,3]
print(mylist[10])
```

Arithmetic Operators

- The addition, subtraction, multiplication, and division operators can be used with numbers

```
number = 1 + 2 * 3 / 4.0
```

- Modulo (%) operator returns the integer remainder of the division

```
remainder = 11 % 3
```

- Using two multiplication symbols makes a power relationship

```
squared = 7 ** 2
cubed = 2 ** 3
```

Using Operators With Strings

- **Python supports concatenating strings using the addition operator**

```
helloworld = "hello" + " " + "world"
```

- **Python also supports multiplying strings to form a string with a repeating sequence:**
  - “hellohellohellohellohellohellohellohellohellohello”:

```
lotsofhellos = "hello" * 10
```

Using Operators with Lists

- **Lists can be joined with the addition operators:**

```
even_numbers = [2,4,6,8]
odd_numbers = [1,3,5,7]
all_numbers = odd_numbers + even_numbers
```

- **Just as in strings, Python supports forming new lists with a repeating sequence using the multiplication operator:**

```
print [1,2,3] * 3
```

Len, Index and Count

- That prints out 12, because "Hello world!" is 12 characters long, including punctuation and spaces

```
print len(astring)
```

- That prints out 4, because the location of the first occurrence of the letter "o" is 4 characters away from the first character.

```
print astring.index("o")
```

- This counts the number of l's in the string. Therefore, it should print 3.

```
print astring.count("l")
```

Conditions

- **Python uses boolean variables to evaluate conditions**
  - variable assignment is done using a single equals operator "="
  - comparison between two variables is done using the double equals operator "=="

```
x = 2
print x == 2 # prints out True
print x == 3 # prints out False
print x < 3  # prints out True
```

Boolean Operators

- The "and" and "or" boolean operators allow building complex boolean expressions:

```
name = "John"
age = 23
if name == "John" and age == 23:
    print "Your name is John, and you are also
      23 years old."
if name == "John" or name == "Rick":
    print "Your name is either John or Rick."
```

The "in" Operator

- **The "in" operator could be used to check if a specified object exists within an iterable object container**
  - such as a list:

```
if name in ["John", "Rick"]:
   print "Your name is either
     John or Rick."
```

Code Blocks

- **Python uses indentation to define code blocks, instead of brackets**

```
if <statement is true>:
    <do something>
    ....
    ....
elif <another statement is true>:
# else if
    <do something else>
    ....
    ....
else:
    <do another thing>
    ....
    ....
```

Code Blocks Example

- Test if x equals to 2

```
x = 2
If x == 2:
    print "x equals two!"
else:
    print "x does not equal to two."
```

Loops

- There are two types of loops in Python, for and while
- **For loops iterate over a given sequence**

```
primes = [2, 3, 5, 7]
    for prime in primes:
        print prime
```

"while" Loops

- **While loops repeat as long as a certain boolean condition is met:**

```
# Prints out 0,1,2,3,4
count = 0
while count < 5:
    print count
    # This is the same as count = count + 1
    count += 1
```

"break" Statement

- **Break is used to exit a for/while loop**

```
#Prints out 0,1,2,3,4
count = 0
while True:
    print count
    count += 1
    if count >= 5:
        break
```

" continue" Statement

- **Continue is used to skip the current block**

```
# Prints out only odd numbers - 1,3,5,7,9
   for x in xrange(10):
       # Check if x is even
       if x % 2 == 0:
           continue
       print x
```

Can We Use "else" Clause for Loops?

- **unlike languages like C,CPP.. we can use else for loops**

```
# Prints out 0,1,2,3,4
#and then it prints count value reached 5

count=0
while(count<5):
    print count
    count +=1
else:
    print "count value reached %d"
      %(count)
```

Functions

- Functions are a convenient way to divide your code into useful blocks
  - allowing us to order our code
  - make it more readable
  - reuse it and save some time
- Also functions are a key way to define interfaces so programmers can share their code (e.g., GitHub)

How do you write functions in Python?

- **Functions in python are defined using the block keyword "def", followed with the function's name as the block's name.**

```
def my_function():
    print "Hello From My Function!"
```

- **Functions may**
  - **also receive arguments (variables passed from the caller to the function).**
  - **return a value to the caller, using the keyword- 'return‘**

```
def sum_two_numbers(a, b):
    return a + b
```

How do you call functions in Python?

- **Write the function's name followed by (), placing any arguments within the brackets**

```
# print a simple greeting
my_function()

# after this line x will hold the value 3!
x = sum_two_numbers(1,2)
```

Classes and Objects

- Objects are an encapsulation of variables and functions into a single entity
  - Classes are essentially a template to create your objects
  - Objects get their variables and functions from classes

Basic Class

- A basic class looks like this

```
class MyClass:
    variable = "blah"
    def function(self):
        print "This is a message inside
          the class."
```

- **To assign the above class(template) to an object you would do the following:**

```
myobjectx = MyClass()
```

Accessing Object Variables

- To access the variable inside of the newly created object "myobjectx" do the following:

```
myobjectx.variable
```

- You can create multiple different objects that are of the same class

```
myobjecty = MyClass()
myobjecty.variable = "yackity"
```

- Then print out both values:

```
# This would print "blah"
print myobjectx.variable
# This would print "yackity"
print myobjecty.variable
```

Accessing Object Functions

- To access a function inside of an object you use notation similar to accessing a variable:

```
myobjectx.function()
```

- The above would print out the message, "This is a message inside the class."

Dictionaries

- **A dictionary is a data type similar to arrays, but works with keys and values instead of indexes**
  - Each value stored in a dictionary can be accessed using a key, which is any type of object
    - a string,
    - a number

    instead of using its index to address it.

Storing Dictionaries

- For example, a database of phone numbers could be stored using a dictionary like this:

```
phonebook = {}
phonebook["John"] = 938477566
phonebook["Jack"] = 938377264
phonebook["Jill"] = 947662781
```

Iterating over Dictonaries

- **Dictionaries can be iterated over**

```
for name, number in phonebook.iteritems():
print "Phone number of %s is %d"
      % (name, number)
```

- **To remove a specified index, use either one of the following notations**

```
del phonebook["John"]
```

or

```
phonebook.pop("John")
```

Modules

- **Modules in Python are simply Python files with the .py extension, which implement a set of functions**

Modules

- **Modules are imported from other modules using the import command.**

```
# import the library
import urllib
# use it
urllib.urlopen(...)
```

- You can check out the full list of built-in modules in the Python standard library
  - https://docs.python.org/2/library/

Writing Modules

- Writing Python modules is very simple.
- To create a module of your own,
  - create a new .py file with the module name,
  - and then import it using the Python file name (without the .py extension) using the import command.

List Comprehensions

- **List Comprehensions is a very powerful tool**
  - **It creates a new list based on another list, in a single, readable line**
- Let's say we need to create a list of integers which specify the length of each word in a certain sentence,
  - but only if the word is not the word "the".

Without List Comprehensions

```
sentence = "the quick brown fox jumps
over the lazy dog"
    words = sentence.split()
    word_lengths = []
    for word in words:
        if word != "the":
            word_lengths.append(len(word))
```

With List Comprehension

- **With a list comprehension, we could simplify this process to this notation:**

```
sentence = "the quick brown fox jumps
     over the lazy dog"
words = sentence.split()
word_lengths = [len(word) for word in
     words if word != "the"]
```

Fixed Function Arguments

- Every function in Python receives a predefined number of arguments, if declared normally, like this:

```
def myfunction(first, second, third):
    # do something with the 3 variables
    ...
```

Multiple Function Arguments

- **It is possible to declare functions which receive a variable number of arguments, using the following syntax:**

```
def foo(first, second, third, *therest):
    print "First: %s" % first
    print "Second: %s" % second
    print "Third: %s" %
    third
    print "And all the rest... %s"
          % list(therest)
```

Output

- So calling foo(1,2,3,4,5) will print out:

```
First: 1
Second: 2
Third: 3
And all the rest... [4, 5]
```

Regular Expressions

- **Regular Expressions (sometimes shortened to regexp, regex, or re) are a tool for matching patterns in text.**
  - The applications for regular expressions are wide-spread, but they are fairly complex

Regular Expressions

- An example regex is r"^(From|To|Cc).*?python-list@python.org“
  - the caret ^ matches text at the beginning of a line
  - the part with (From|To|Cc) means that the line has to start with one of the words that are separated by the pipe |.
    - That is called the OR operator, and the regex will match if the line starts with any of the words in the group.
  - The .*? means to match any number of characters, except the newline \n character.

Example

- So, the following lines would be matched by that regex:
  - From: python-list@python.org
  - To: !asp]<,. python-list@python.org
- A complete reference for the re syntax is available at the python docs:
  - https://docs.python.org/3/library/re.html#regular-expression-syntax "RE syntax

Exception Handling

- When programming, errors happen.
  - It's just a fact of life...
    - Perhaps the user gave bad input..
    - Maybe a network resource was unavailable.. Maybe the program ran out of memory..
    - Or the programmer may have even made a mistake!

Exceptions

- Python's solution to errors are exceptions

```
>>> print a
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Try/Exceptions Handling

- **But sometimes you do not want exceptions to completely stop the program.**
  - You might want to do something special when an exception is raised
  - **This is done in a *try/except* block**

Example

- Suppose you're iterating over a list.
  - You need to iterate over 20 numbers, but the list is made from user input, and might not have 20 numbers in it
  - After you reach the end of the list, you just want the rest of the numbers to be interpreted as a 0

Exception Handling Code

```
def do_stuff_with_number(n):
    print n

the_list = (1, 2, 3, 4, 5)

for i in range(20):
   try:
       do_stuff_with_number(the_list[i])
   # Raised when accessing
   # a non-existing index of a list
   except IndexError:
      do_stuff_with_number(0)
```

Sets

- **Sets are lists with no duplicate entries.**

```
print set("my name is Eric and Eric is
      my name")
```

- This will print out a list containing "my", "name", "is", "Eric", and finally "and".
  - Since the rest of the sentence uses words which are already in the set, they are not inserted twice.

Operations over Sets

- **Sets allow to calculate intersections and differences**
  - For example, say:

```
a = set(["Jake", "John", "Eric"])
b = set(["John", "Jill"])
```

- **To find out which members attended both events, you may use the "intersection" method:**

```
>>> a.intersection(b)
set(['John'])
```

Difference and Union

- **To find out which members attended only one of the events, use the "symmetric_difference" method:**

```
>>> a.symmetric_difference(b)
set(['Jill', 'Jake', 'Eric'])
```

- **To find out which members attended only one event and not the other, use the "difference" method:**

```
>>> a.difference(b)
set(['Jake', 'Eric']
```

- **To receive a list of all participants, use the "union" method:**

```
>>> a.union(b)
set(['Jill', 'Jake', 'John', 'Eric'])
```

Partial Functions

- **You can create partial functions in python by using the partial function from the functools library**
  - **Partial functions allow to derive a function with x parameters to a function with fewer parameters**
  - Import required:

```
from functools import partial
```

Example

- Example: from functools import partial
  - ```
    from functools import partial

    def multiply(x,y): return x * y
    # create a new function
    # that multiplies by 2
    dbl = partial(multiply,2)
    print dbl(4)
    ```
- This code will return 8.
- An important note:
  - the default values will start replacing variables from the left. The 2 will replace x. y will equal 4 when dbl(4) is called.
    ```

Reading List

- **Review Slides**
- **Recommended**
  - Python Tutorial
    - http://www.learnpython.org/
- **Optional**
  - History and General Information
    - https://en.wikipedia.org/wiki/History_of_Python
    - https://www.python.org/doc/essays/comparisons
    - http://www.programmerinterview.com/index.php/general-miscellaneous/whats-the-difference-between-a-compiled-and-an-interpreted-language/
  - Documentation
    - https://docs.python.org/2/tutorial/