Python Fundamentals

Scripts are Usually Interpreted

- Using an interpreter instead of a compiler makes sense when programs change frequently and/or are very interactive.
  - Reason: no extra compile time
- In the scripting language-as-glue-language mode, performance is dominated by the modules being connected by the scripting commands

Why Python?

- No universal agreement on which scripting language is best
- Perl, Ruby, Python all have advocates
- Perl: good for system administration duties, traditional scripting
- Python – has attracted many former Perl users
- Ruby – syntactically similar to Perl and Python; popularized because it was used to write Ruby on Rails, a framework for web apps.

Python - Intro

- Python is a general purpose scripting language that implements the imperative, object-oriented, and functional paradigms.
- Dynamic typing, automatic memory management, exceptions, large standard library, modular.
  - Extensions can be written in C and C++
  - Other language versions (Jython, IronPython) support extensions written in Java and .Net languages)
- Design philosophy: easy to read, easy to learn

Versions

- Current production versions are 2.6.4 and 3.1.1
- Both versions are stable and suitable for use
- Python 2: compatible with much existing software
- Python 3: a major redesign
  - Not backward compatible.
  - Most features are the same or similar, but a few will cause older programs to break.
  - Part of the Python philosophy – don’t clutter up the language with outdated features

Interesting Features

- White space does indicate meaning
  - Instead of curly brackets or begin-end pairs, whitespace is used for block delimiters.
  - Indent statements in a block, un-indent at end of block.
- Statements are terminated by `<Enter>`
- No variable declarations
- Dynamic typing
- Associative arrays (dictionaries)
- Lists and slices

Execution Modes - Calculator Mode

Type in an expression Python will evaluate it

```
>>> a = 5
>>> b = 10
>>> a + b
15
>>> x = a + b
>>> x
15
```

Dynamic type change

```
>>> a = 'horse’
>>> b = ‘ cart’
>>> a + b
‘horse cart’
>>> a + x
Traceback (most recent
call last):
    File
  "<pyshell#10>",line
  1,in <module>
    a + x
  TypeError: Can't
  convert 'int' object to
  str implicitly
```

Execution Modes - Program

```
>>> #factorial.py
>>> #compute factorial
>>> def main( ):
  n = int(input("enter an int "))
  fact = 1
  for factor in range (n, 1, -1):
     fact = fact * factor
  print("the answer is: ", fact)


>>> main( )
enter an int 5
the answer is:  120
>>>
```

Execution Modes - Program

- You can also create the program in a text file, adding the statement

  ```
  main()
  ```

  as the last statement, and then run it later.

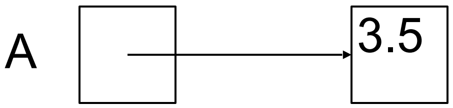
Program Elements

- Identifiers:
  - Must begin with letter or underscore, followed by any number of letters, digits, underscores
- Variables:
  - Do not need to be declared
  - A variable is created when a value is assigned to it: Examples: `num = 3`
  - Can’t be used in an expression unless it has a value
  - Error message: *Name Error* – means no value is associated with this name

Variables

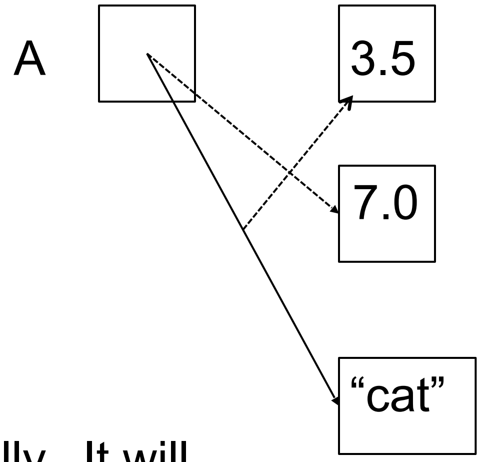
- Variable names don’t have static types – values (or objects) do
  - A variable name has the type of the value it currently references
- Variables actually contain references to values (similar to pointers).
  - This makes it possible to assign different object types to the same variable

Variables contain references to data values



A = A * 2

A = “cat”



Python handles memory management automatically. It will create new objects and store them in memory; it will also execute garbage collection algorithms to reclaim any inaccessible memory locations.

Python does not implement reference semantics for simple variables; if A = 10 and B = A, A = A + 1 does not change the value of B

Basic Data Types

- Numeric types: integer, floats, complex
- A literal with a decimal point is a float; otherwise an integer
- Complex numbers use “j” or “J” to designate the imaginary part: `x = 5 + 2j`
- `type()` returns the type of any data value:

Data Types

```
>>> type(15)
<class 'int'>
>>> type (3.)
<class 'float'>
>>> x = 34.8
>>> type(x)
<class 'float'>
```

```
>>> 1j * 1j
(-1+0j)
>>> s = 3 + 1j
>>> type(s)
<class 'complex'>
>>> x = "learning”
>>> type(x)
<class 'str'>
```

Expressions

- An expression calculates a value
- Arithmetic operators: +, -, *, /, ** (exponentiation)
- Add, subtract, multiply, divide work just as they do in other C-style languages
- Spaces in an expression are not significan

Expressions

- Mixed type (integer and float) expressions are converted to floats:

```
>>> 4 * 2.0 /6
1.3333333333333333
```

- Mixed type (real and imaginary) conversions:

```
>>> x = 5 + 13j
>>> y = 3.2
>>> z = x + y
>>> z
(8.2 + 13J )
```

- Explicit casts are also supported:

```
>>> y = 4.999                          >>> x = 8
>>> int(y)                             >>> float(x)
4                                     8.0
```

Assignment Statements

- Syntax: A*ssignment* → *variable* = *expression*
- A variable's type is determined by the type of the value assigned to it.
- Multiple_assign →*var{, var}* = *expr{, expr)*

```
>>> x, y = 4, 7
>>> x
4
>>> y
7
>>> x, y = y, x
>>> x
7
>>> y
4
>>>
```

Interactive Input

- Syntax: *input → variable = input(string)*
  The string is used as a prompt.
  Inputs a string

```
>>> y = input("enter a name --> ")
enter a name --> max
>>> y
'max'
>>> number = input("Enter an integer ")
Enter an integer 32
>>> number
’32’
```

- *input()* reads input from the keyboard as a string;

Input

- To get numeric data use a cast :

```
>>> number = int(input("enter an integer: "))
enter an integer: 87
>>> number
87
```

- If types don’t match (e.g., if you type 4.5 and try to cast it as an integer) you will get an error:

```
ValueError: invalid literal for int() with base 10:
```

Input

- Multiple inputs:

```
>>> x, y = int(input("enter an integer: ")),
                float(input("enter a float:
  "))
enter an integer: 3
enter a float: 4.5
>>> print("x is", x, " y is ", y)
x is 3  y is  4.5
```

- Instead of the cast you can use the *eval( )* function and Python choose the correct types:

```
>>> x, y = eval(input("Enter two numbers: "))
Enter two numbers: 3.7, 98
>>> x, y
(3.7, 98)
```

Python Control Structures

- Python loop types:
  - while
  - for
- Decision statements:
  - if
- Related features:
  - range( )   # a function
  - break   # statements similar to
  - continue   # those in C/C++

General Information

- The control structure statement must end with a semicolon (:)
- The first statement in the body of a loop must be indented.
- All other statements must be indented by the same amount
- To terminate a loop body, enter a blank line or “unindent”.

While Loop

```
>>> x = 0
>>> while (x < 10):  # remember semicolon!
        print(x, end = " ")#no new line
        x = x + 1

0 1 2 3 4 5 6 7 8 9
```

Conditions are evaluated just as in C/C++: 0 is false, non-zero is true

The conditional operators are also the same

Note indentation – provided by the IDLE GUI

While Loop

```
for x in range(10):
    # This is in the loop
    print("We are currently on the number {}!".format(x))

    if x == 5:
        # This is in the if statenent which is in the loop
        print("Cool! 5!")

    if x % 2 == 0:
        # This is in a different if statement but still in the loop
        print("Alright! {} is an even number!".format(x))

    # This is in the for loop but not in a if statement
    print("Next number coming up... {}!".format(x+1))

# This code isn't in any kind of block
print("Finished!")
```

For Loops

- Syntax:

```
for <var> in <sequence>:
   <body>
```

- *<sequence>* can be a list of values or it can be defined by the `range( )` function
  - range(n) produces a list of values: 0, 1, …,n-1
  - range(start, n): begins at start instead of 0
  - range(start, n, step): uses step as the increment

```
>>> for i in range(3):
          print(i,end = " ")
0 1 2
>>> for i in range(5,10):
           print(i,end = " ")
5 6 7 8 9
>>> for i in range(6,12,2):
           print(i)
6
8
10
>>> for i in (1, 2, 3):
     print(i)
1
2
3
```

Ranges can also be specified using expressions:

```
>>> n = 5
>>> for i in range(2*n + 3):
        print(i,end = “ “)


0 1 2 3 4 5 6 7 8 9 10 11 12
```

Using a List in a `for` loop

Lists are enclosed in square brackets

```
>>> for i in [3, 2, 1]:
         print(i,end = " ")

3 2 1
>>> for i in ['cat', 'dog', 'elephant']:
         print(i,end = " ")

cat dog elephant
```

If Statement

- Python `if` statement has three versions:
  - The `if` (only one option)
  - The `if-else` (two options)
  - The `if-elif-else` (three or more options)
- The `if-elif-else` substitutes for the `switch` statement in other languages.
- Each part must be followed with a semicolon and whitespace is used as the delimiter.

If-elif-else Statement

```
x = int(input("enter an integer: “))
if x < 0:
       print (‘Negative’)
elif x == 0:
    print ('Zero‘)
elif x == 1:
    print ('Single')
else:
    print ('More' )
```

Right way

```
>>> if x < 0:
            y = x
elif x == 0:
            y = 1
elif x < 10:
            y = 100
else:
            print("none”)
```

(you may have to override the IDLE indentation)

Wrong way

```
>>> if x < 0:
            y = x
    elif x == 0:
            y = 1
    elif x < 10:
            y = 100
    else:
            print("none ")
```

```
SyntaxError: unindent does
   not match any outer
   indentation level
   (<pyshell#86>, line 3)
```

Python Data Types/Data Structures

- Many built-in simple types: ints, floats, infinite precision integers, complex, string, etc.
- Built-in data structures:
  - Lists or dynamic arrays
  - Dictionaries (associative arrays or hash tables)
    - Accessed by key-value indices
  - Tuples
    - Similar to lists, but cannot be modified
    - Sometimes used like structs, but indexed by position instead of field name

String Data Type

- A sequence of characters enclosed in quotes (single or double; just be consistent)
- Elements can be accessed via an index, but you cannot change the individual characters – you will get an error if you try:
- `TypeError: 'str' object does not support item assignment`

String Data Type

```
>>> str1 = 'happy'
>>> str2 = "Monday"
>>> str1, str2
(‘happy', 'Monday')  # a tuple
>>> str1[1]
'a'
>>> x = str1, str2
>>> x
('happy', 'Monday')  # x is a tuple
>>> x[1]
'Monday'
```

String Data Type - continued

```
>>> print(str1, str2)
Happy Monday
>>> "I don't like hotdogs"
"I don't like hotdogs"
>>> 'I don't like hotdogs'
>>>
Syntax Error: invalid syntax
```

Use double quotes when you want to include single quotes (and vice versa)

String Data Type

- More examples:

```
>>> '"why not?" said Jim'
 '"why not?" said Jim'
 >>> 'can\'t'
 "can't“
```

You can get the same effect by using the escape sequence \’ or \”

String Operations

- Indexing: 0-based indexing for string characters
- General format: `<string>[<expr>]`

```
>>> name = 'bob jones'
>>> name[0]
  'b'
  >>> x = 3
  >>> name[x + 2]
  ‘o’
```

- Negative indexes work from right to left:

```
>>> myName = "Joe Smith"
>>> myName[-3]
'i'
```

String Operations

- Slicing: selects a ‘slice’ or segment of a string *<string>[<start>:<end>]* where *<start>* is the beginning index and *<end>* is one past final index

```
 >>> myName = "Joe Smith"
 >>> myName[2:7]
'e Smi`
```

- If either *<start>* or *<end>* is omitted, the start and end of the string are assumed

```
 >>> myName[:5]
 'Joe S'
 >>> myName[4:]
 'Smith'
 >>> myName[:]
 'Joe Smith'
```

String Operations

Concatenation (+)

```
 >>> "happy" + "birthday"
'happybirthday'
 >>> 'happy' + ' birthday'
 'happy birthday’
```

Repetition (*)

```
>>> word = 'ha'
>>> 3 * word
'hahaha'
>>> word + 3 * '!'
'ha!!!'
```

String Operations

Other examples:

```
>>> word = 'ha'
>>> len(word)  # length function
2
>>> len("ham and eggs")
12
>>> for ch in myName:
          print(ch, end = " ")


J o e   S m i t h
```

Lists

- A list is a comma-separated sequence of items, enclosed in square brackets
- Lists can be heterogeneous – items don’t have to be from the same data type
- Like strings, lists can be sliced, indexed, concatenated, and repeated.
- The `len()` function will return the number of elements in the list

Lists

- Unlike strings, lists are mutable (can be changed by element assignment)
  - Make an assignment, using the index

```
>>> myList = ['milk','eggs','bread']
>>> myList[1] = 'butter'
>>> myList
['milk', 'butter', 'bread']
```

- You can also assign to slices, and even change the length of the list

List Slice Operations

```
#create a list and assign to a variable
>>> data = ['bob', 32, 'sue', 44]
>>> data
['bob', 32, 'sue', 44]
#assign to a list slice
>>> data[1:3] = ['dave', 14]
>>> data
['bob', 'dave', 14, 44]
#insert an element (or several)
>>> data[1:1] = [19]
>>> data
['bob', 19, 'dave', 14, 44]
#delete an element
>>> data[3:4] = []
>>> data
['bob', 19, 'dave', 44]
```

Python Lists

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[1] = 6
>>> b
[1, 6, 3]
>>> a = [6, 7, 8]
>>> b[2] = 'x'
>>> b
[1, 6, 'x']

>>> # copy or reference semantics?
```

Python Lists

```
>>> x = 13
>>> L1 = [x, 'y', 3]
>>> L1
[13, 'y', 3]
>>> x = 19
>>> L1
[13, 'y', 3]
```

Adding to a List

- You can grow the list dynamically with the concatenation operator:

```
>>> x = [2, 4, 6, 8]
>>> x
[2, 4, 6, 8]
>>> x = x + [10]
>>> x
[2, 3, 6, 8, 10]
```

List Insertion by Slicing

```
>>> x = [2,3,6,8.10]
>>> x
[2, 3, 6, 8.1]__________________
>>> x = x[:2]+[100,200] + x[2:]
>>> x
[2, 3, 100, 200, 6, 8.1]
>>> x = x[:4] + []
>>> x
[2, 3, 100, 200]
```

Nested Lists

```
>>> grades = [100, 97, 85]
>>> stRec = ['A000','jack',grades]
>>> stRec
['A000', 'jack', [100, 97, 85]]
>>> len(stRec)
3
```

Additional String Operations

- `split`: divides a string into a list of substrings

```
 >>> myStr = 'The fat black cat'
  >>> myStr.split()
   ['The', 'fat', 'black', 'cat']
```

- `split` defaults to blank as the delimiter, but you can specify a different character:

```
>>> myStr = '12/10/2008'
>>> myStr.split('/')
['12', '10', '2008']
```

Strings

- After you have split a string into a list of substrings, you may want to convert some of the substrings to specific data types.
  - specific casts: *int( ), float( ), long( ),* and str( )
  - If you don't know the data types, you can use the generic cast *eval( )*

Example

```
>>> mysplitStr
['12', '10', '2008']
>>> first = eval(mysplitStr[0])
>>> first
12
>>> #etc. - you can use the type function to
  determine if the list elements are the
  type you expected:
>>> x = 3.4
>>> if type(x) == float:
  print('float')

float
```

More About Lists

- See Python tutorial to get a whole set of list functions:
  - append(x)
  - insert(i, x)
  - etc.
- Since lists are objects, dot notation is used to call the functions
- When using string functions you may need to import the string library:
  - **`import string`**

List Functions

```
>>> stRec = ['A000', 'jack', grades]
>>> stRec.remove(grades)
>>> stRec
['A000', 'jack']
>>> stRec.append([100, 97, 85])
>>> stRec
['A000', 'jack', [100, 97, 85]]
>>> stRec.pop(2)#removes item at index
[100, 97, 85]
>>> stRec
['A000', 'jack']
```

Python Tuples
5.3 in Tutorial

- A tuple is a sequence of comma-separated values:

```
>>> t =('A000','jack',3.56,'CS')
```

- Tuples can be indexed like lists and strings.
- Like strings, tuples are immutable (cannot assign to individual elements)
- Tuples can be sliced and concatenated

Tuple Example

```
>>> t = ('A000', 'jack', 3.56, 'CS')
>>> t1 = ('A001', 'jill', 2.78, 'MA')
>>> t2 = ('A0222', 'rachel', 3.78, 'CS')
>>> students = [t, t1, t2]
>>> for (ID, name, GPA, major) in students:
          print(ID, name, GPA, major)

A000 jack 3.56 CS
A001 jill 2.78 MA
A0222 rachel 3.78 CS
>>>
```

For each iteration, one tuple in the list of tuples is unpacked into the individual variables

Sequences

- Strings,lists, and tuples are all examples of the *sequence* data type.
- Operations that can be performed on sequences:

<seq> + <seq>	concatenation
<seq> * <int-exp>	repetition
<seq>[ ]	indexing
<seq>[:]	slicing
len(<seq>)	length
for <var> in <seq>	iteration

- Only the list sequence type is modifiable

Dictionaries

- A dictionary is an example of a *mapping* object.
- Some languages call them associative arrays or hashes.
- Unlike sequences (lists, strings, tuples) which are indexed by an ordered range of values, dictionaries are indexed by *keys*
- Items are retrieved according to their keys.

Dictionary Entries

- Dictionaries contain <key, value> pairs
- The key is a unique identifier (student #, account #, SSN, etc.) and the value is whatever data might be associated with the key; e.g., name, address, age, …
- The value can be one thing or it can be a list, or sequence, or tuple, or …

Dictionaries

- A key must be unique within a given dictionary object.
- Keys must be hashable; i.e., they cannot contain lists, dictionaries, or other mutable objects. Tuples can be used as keys as long as they don’t contain lists, dictionaries, …
- Items are retrieved according to their keys.

Dictionaries

- List: an ordered collection of elements
- Dictionary: an unordered collection of elements (items aren’t stored sequentially)
- Other characteristics:
  - Mutable (add, delete elements or change them)
  - Variable length
  - Cannot be sliced or concatenated

Creating Dictionaries

Create an empty dictionary:

```
>>> roll = { }
```

Create and initialize a dictionary:

```
>>> roll = {1023: 'max', 404: 'sue'}
```

Retrieve an item by its key:

```
>>> roll[1023]
'max'
```

Add an item:

```
>>> roll[9450] = 'alice'
>>> roll
{9450:'alice', 404:'sue', 1023:'max'}
```

Alternate Dictionary Creation

Using the dict() function:

```
>>> tel = dict(max = (94, 'x'),
  alice = (5, 'y'))
>>> tel
{‘max': (94, 'x'), ‘alice': (5,
  'y')}
```

To get a list of the keys:

```
>>> list(roll.keys())
[9450, 404, 1023]
```

To get the list in sorted form:

```
>>> x = sorted(roll.keys())
>>> x
[404, 1023, 9450]
```

To find out if a key is in the dictionary:

```
>>> 2600 in roll
False
```

To remove a key

```
>>> del roll[404]
>>> roll
{9450: 'alice', 1023: 'max'}
```

To change an element’s value:

```
>>> roll[9450] = ('alice', 3.45, 'CS')
>>> roll
{9450:('alice', 3.45, 'CS'), 1023:'max'}
```

Sort on the value field if sortable:

```
>>> numd = {34: 56, 45:100, 906: 25, 100: 3}
>>> numd
{34: 56, 100: 3, 45: 100, 906: 25}
>>> sorted(numd.values())
[3, 25, 56, 100]
>>> sorted(numd.keys())
[34, 45, 100, 906]
```

Looping Through Dictionaries

You can retrieve the key and the value at the same time:

```
>>> d = dict(max = 37, joe = 409, cal = 100)
>>> for name, code in d.items():
          print(name, code)

max 37
cal 100
joe 409
```

Notice the items are processed in stored order, not the order in which they were entered

Python Functions

- A Python function can be either void or value returning, although the definition isn't identified as such.
  - Technically, they are all value returning, since a void function returns the value **`none`**
- Each function defines a scope. Parameters and local variables are accessible in the scope; values in other functions are available only as parameters.

Function Syntax

Function definition

```
def <name> (formal-parameters>):
    <body>
```

Function call syntax:

```
<name>.(<actual parameters>)
```

```
>>> def square(x):
     return x * x

>>> square(10)
100
>>> z = 23
>>> square(z * 4)
8464
>>> x = square(z)
>>> x
529
```

You can enter functions at the command line and use in calculator mode, or include them in programs, or in modules (similar to libraries).

A main program file

```
#File: chaos.py
#A simple program illustrating chaotic behavior

def main( ):
    print("a chaotic function“)
    x=eval(input("enter a number betw 0 and 1: ")
    for i in range (10):
        x = 3.9 * (1 - x)
        print(x)
    # freeze the output window:
    y = input("Press any key to exit ")

main()
```

Functions That Return Values

```
>>> def sum(x, y):
      sum = x + y
      return sum

>>> num1,num2 = 3,79
>>> sum(num1, num2)
82
```

```
>>> def SumDif(x,y):
      sum = x + y
      dif = x - y
      return sum, dif

>>> x, y = 7, 10
>>> a,b = SumDif(x,y)
>>> a
17
>>> print(a, b)
17 -3
>>>
```

```
>>> def incre(list):
    n = len(list)
    for i in range(n):
        print(i)
        list[i] = list[i] + 1

>>> myList = [2, 4, 6]
>>> incre(myList)
0
1
2
>>> print(myList)
[3, 5, 7]
>>>
```

File I/O

- To open a file:
  `<filevar> = open(<filename>, <mode>)`
  where **`filename`** is the name of the file on disk, and **`mode`** is usually **`‘r’`** or **`‘w’`**
  - **`infile = open(“data.in”, “r”)`**
  - **`outfile = open(“data.out”, “w”)`**
- Files must also be closed:
  - **`infile.close( )`**
  - **`outfile.close( )`**

Input Operations

- **`<filevar>.read`**: returns the remaining contents of the file as a multi-line string (lines in the file separated by \n)
- **`<filevar>.readline( ):`** returns next line as a string; includes the newline character (you can slice it off, if needed)
- **`<filevar>.readlines( ):`** returns the remaining lines in the file, as a list of lines (each list item includes the newline character)

Detecting End of File (from the documentation)

- f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn’t end in a newline. This makes the return value unambiguous;
- if f.readline() returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.

Detecting End of File (from the documentation)

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
` `
```

Reading a File Line-by-Line

```
>>> infile=open("C:\Temp\Test.txt",'r')
>>> for line in infile:
        myString = line
        print(myString)
This is the first line of the file.
Second line of the file
```

Stops when there are no more lines in the file

File Output

`f.write(string)` writes the contents of *string* to the file, *returning the number of characters written.*

```
>>> f.write('This is a test\n') 15
```

To write something other than a string, convert it to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s) 18
```

Modules

- Modules are additional pieces of code that further extend Python's functionality
- A module typically has a specific function
  - additional math functions, databases, network…
- Python comes with many useful modules
- *arcgisscripting* is the module we will use to load ArcGIS toolbox functions into Python

Modules

- Modules are accessed using import
  - import sys, os # imports two modules
- Modules can have subsets of functions
  - os.path is a subset within os
- Modules are then addressed by modulename.function()
  - sys.argv # list of arguments
  - filename = os.path.splitext("points.txt")
  - filename[1] # equals ".txt"

Modules

Save this code in the file `mymodule.py`

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Import the module named mymodule, and access the person1 dictionary:

```
import mymodule

a = mymodule.person1["age"]
print(a)
```

Built-in Functions & Modules

https://www.w3schools.com/python/python_reference.asp