ESCI 386 – Scientific Programming, Analysis and Visualization with Python

Lesson 2 – Syntax and Data Types

Comments

- Comments in Python are indicated with a pound sign, #.
- Any text following a # and before the next carriage return is ignored by the interpreter.
- For multiple-line comments a # must be used at the beginning of each line.

Continuation Line

- The \ character at the end of a line of Python code signifies that the next line is a continuation of the current line.

Variable Names and Assignments

- Valid characters for variable, function, module, and object names are any letter or number. The underscore character can also be used.
- Numbers cannot be used as the first character.
- The underscore should not be used as either the first or last character, unless you know what you are doing. There are some special rules concerning leading and trailing underscore characters.

Variable Names and Assignments

- Python is case sensitive! Capitalization matters.
  - The variable m and the variable M are not the same.

- Python supports parallel assignment

```
>>> a, b = 5, 'hi'
>>> a
5
>>> b
'hi'
```

Data Types

- Examples of *data types* are integers, floating-point numbers, complex numbers, strings, etc.

- Python uses *dynamic typing*, which means that the variable type is determined by its input.
  - The same variable name can be used as an integer at one point, and then if a string is assigned to it, it then becomes a string or character variable.

Numeric Data Types

- Python has the following numeric data types
  - Boolean
  - Integer
  - Floating-point
  - Complex

Boolean Data Type

- The *Boolean* data type has two values: True and False (note capitalization).
  - True also has a numerical value of 1
  - False also has a numerical value of 0

```
>>> True == 1
True
>>> True == 2
False
>>> False == 1
False
>>> False == 0
True
```

Integer Data Type

- There are two integer data types in Python:
  - *Integer*
    - Ranges from approximately –2147483648 to +2147483647
    - Exact range is machine dependent
  - *Long integer*
    - Unlimited except by the machine's available memory

Integer Data Type

- That there are two types of integers is pretty transparent to the user (a long integer is denoted by having an L after the number.)

```
>>> a = 34
>>> a
34
>>> b = 34*20000000000000000000
>>> b
680000000000000000000L
```

Floating-point Data Type

- All floating-point numbers are 64 bit (double-precision)
- Scientific notation is the same as in other languages
  - Either lower or upper case (e or E) can be used.

```
>>> a = 67000000000000000000.0
>>> a
6.7e+19
>>> b = 2E3
>>> b
2000.0
```

Complex Data Type

- Complex numbers such as 4.5 + *i*8.2 are denoted 4.5 + 8.2j
  - Either lower-case or upper-case j or J may be used to denote the imaginary part.
  - The complex data type has some built-in attributes and methods to retrieve the real part, the imaginary part, and to compute the conjugate:

Complex Data Type Example

```
>>> c = 3.4 + 5.6j
>>> c
(3.4+5.6j)
>>> c.real
3.4
>>> c.imag
5.6
>>> c.conjugate()
(3.4-5.6j)
```

Objects, Attributes, and Methods

- The complex number example above provides an opportunity to discuss the object-oriented nature of Python.

- In Python most entities are *objects*. In the example, the complex number c is an object that represents an *instance* of the complex *class*.

Attributes

- Objects may have *attributes* associated with them.
  - The attributes can be thought of as some type of data that is bound to the object.
  - Each attribute has a name.
  - The value of the attribute is found by typing the name of the object, a period, and then the name of the attribute, in the form object.attribute

Attributes of the Complex Class

- In the complex number example, the complex class has two attributes named ‘real’ and ‘imag’ that return the real and imaginary parts of the complex number.
  - The command c.real accessed the attribute named ‘real’ of the complex number c.
  - Likewise the command c.imag accessed the attribute named ‘imag’.

Methods

– A method can be thought of as a function that belongs to the object, and operates on the objects attributes, or on other arguments supplied to the method.

– An object’s methods are invoked by typing the name of the object, a period, and then the name of the method, along with parenthesis for the argument list, in the form object.method(*[…argument list…]*)
  - Note: The parenthesis must always be present to invoke a method, even if there are not arguments needed.

Methods of the Complex Class

- In the complex number example, the complex class has a method called conjugate() that returns the conjugate of the number represented by the object.
  - In the example there are no arguments that need to be passed to the method.

The None Data Type

- An object or variable with no value (also known as the *null value*) has data type of None (note capitalization).

- A value of None can be assigned to any variable or object in the same manner as any other value is assigned.

```
>>> a = None
>>> a
>>>
```

Strings

- The string data type is assigned by enclosing the text in single, double, or even triple quotes. The following are all valid ways of denoting a string literal
  - ‘Hello there’
  - “Hello there”
  - ‘‘‘Hello there’’’
  - “““Hello there”””

Mixing Quotes

- Mixing single, double, and triple quotes allows quotes to appear within strings.

```
>>> s = 'Dad said, "Do it now!"'
>>> s
'Dad said, "Do it now!"‘
>>> print(s)
Dad said, "Do it now!"
```

Triple Quotes

- Triple-quoted strings can include multiple lines, and retain all formatting contained within the triple quotes.

```
>>> s = '''This sentence runs
 over a
 few lines.'''
>>> s
'This sentence runs\n over a\n few lines.'
>>> print(s)
This sentence runs
 over a
 few lines.
```

Special Characters

- Special characters within string literals are preceded by the backslash, \
- One common special character is the newline command, \n, which forces any subsequent text to be printed on a new line.

```
>>> print('Hello \n there.')
Hello
 there.
```

Escaping Special Characters

- The backslash character can be escaped by preceding it with another backslash.

```
>>> print('Hello \\n there.')
Hello \n there.
```

Escaping Quotes and Apostrophes

- The backslash can also be used to escape quotes and apostrophes within a string.

```
>>> s = 'An apostrophe looks like \'.'
>>> print(s)
An apostrophe looks like '.
```

Raw Strings

- Raw strings are strings where backslashes are left as is, and are not interpreted as escape characters.

- Raw strings are defined by preceding the string literal with r.

```
>>> s = r'Hello \n there\'.'
>>> print(s)
Hello \n there\'.
```

Formatting Strings

- Strings are formatted in Python using the .format() method.

```
>>> x = 654.589342982
>>> s = 'The value of x is {0:7.2f}'.format(x)
>>> s
'The value of x is  654.59'
```

Formatting Strings (cont.)

- The curly braces {} indicate that we will be adding values into the string using the .format() method.

- Within the curly braces the syntax is {*n*:*fs*} where *n* is the *n*th variable in the argument to the format() method, and *fs* is as format specifier detailing how the value is to appear.

Format Specifiers

Spec.	Explanation	Examples
d	integer	`'{0:d}'.format(45) => '45'`
*w*d	field width of *w*	`'{0:5d}'.format(45) => '   45'`
+*w*d	force sign to be included	`'{0:+5d}'.format(45) => '  +45'`
0*w*d	pad with zeros	`'{0:05d}'.format(45) =>'00045'`
f	floating point	`'{0:f}'.format(-3.5) => '-3.500000'`
w.*d*f	field width *w* and *d* decimal places	`'{0:6.2f}'.format(-3.5) => ' -3.50'`
0*w*.*d*f	pad with zeros	`'{0:06.2f}'.format(-3.5) => '-03.50'`
+*w*.*d*f	force sign to be included	`'{0:+6.2f}'.format(-3.5) => ' +3.50'`
e	scientific notation	`'{0:e}'.format(0.654) => '6.540000e-01'`
w.*d*e	field width *w* and *d* decimal places	`'{0:9.2e}'.format(0.654) => ' 6.54e-01'`
+*w*.*d*e	force sign to be included	`'{0:+9.2e}'.format(0.654) => '+6.54e-01'`

Format Specifiers

Spec.	Explanation	Examples

g	uses scientific notation for
	exponents less than –4.
		`'{0:g}'.format(45679.3) => '45679.3'`
		`'{0:g}'.format(0.00346) => '0.00346'`
		`'{0:g}'.format(0.0000346) => '3.46e-05'`

%
w.d%
0*w.d*%
	converts decimals to percent
	field width *w* and *d* decimal places
	pad with zeros
		`'{0:%}'.format(0.4567) => '45.670000%'`
		`'{0:8.2%}'.format(0.4567) => '  45.67%'`
		`'{0:8.2%}'.format(0.4567) => '0045.67%'`

s *w*s	string field width of *w*	`'{0:s}'.format('Hello') => 'Hello'` `'{0:9s}'.format('Hello') => 'Hello    '`

Another Example

```
>>> name = 'Tom Jones'
>>> age = 45
>>> weight = 177.8
>>> height = 70.32
>>> s = 'His name is {0:s}.  His is {1:d} years old. He is {2:.0f} inches tall and weighs {3:.1f} pounds.'
>>> s.format(name, age, height, weight)
'His name is Tom Jones.  His is 45 years old. He is 70 inches tall and weighs 177.8 pounds.'
```

Lists and Tuples

- Lists and tuples are both collections of values or objects.
  - The data types of the objects within the list do not have to be the same.

- Lists are denoted with square brackets, while tuples are denoted with parentheses.

```
>>> l = [4.5, -7.8, 'pickle', True, None, 5]    ← list
>>> t = (4.5, -7.8, 'pickle', True, None, 5)    ← tuple
```

Tuples versus Lists

- Lists can be modified after they are created.
  - Lists are *mutable*.

- Tuples cannot be modified after they are created.
  - Tuples are *immutable*

Lists and Tuples May Contain Other Lists and Tuples

```
>>> l = [4.5, ('cat', 'dog'), -5.3, [4, 8, -2], True]
```

Accessing Lists and Tuples

- The individual elements of a list or tuple are accessed by denoting their indices within square brackets.

```
>>> t = [0,-5, 8, 'hi', False]
>>> t[0]
0
>>> t[1]
-5
>>> t[2]
8
>>> t[3]
'hi'
>>> t[4]
False
```

Can Use Negative Indices

```
>>> t[-1]
False
>>> t[-2]
'hi'
>>> t[-3]
8
>>> t[-4]
-5
>>> t[-5]
0
```

Using Ranges

- Ranges of indices can also be used. These are indicated by the form *start:end*.

- IMPORTANT! The last value in the range is NOT returned.

```
>>> t
[0, -5, 8, 'hi', False]
>>> t[1:3]
[-5, 8]
>>> t[0:-1]
[0, -5, 8, 'hi']
```

Using Ranges

- All the elements from the first up to a given index (minus one) are accessed by starting with a colon.
- All elements from a starting element to the end are accessed by ending with a colon.

```
>>> t
[0, -5, 8, 'hi', False]
>>> t[:4]
[0, -5, 8, 'hi']
>>> t[2:]
[8, 'hi', False]
```

Striding

- Can specify a stride to skip elements.
- A negative stride can move backwards.

```
>>> t = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>> t[0:-1:3]
[1, 4, 7, 10]
>>> t[10:2:-2]
[11, 9, 7, 5]
```

Reversing Elements

- A negative stride can move backwards through list.
- To return all the elements of an array in

```
>>> t = [0,-5,8,'hi',False]
>>> t
[0, -5, 8, 'hi', False]
>>> t[::-1]
[False, 'hi', 8, -5, 0]
```

Accessing Nested Elements

- Nested elements are accessed by multiple indices.

```
>>> n = [[2,3,7], [-2, -4, 8], ['pickle', 'Henry']]
>>> n[0]
[2, 3, 7]
>>> n[0][1]
3
>>> n[2][0]
'pickle'
>>> n[1][1:]
[-4, 8]
```

Assigning/Reassigning Elements

- Since lists are mutable we can reassign values to their elements.

```
>>> p = ['cat', 'dog', 'ferret', 'llama']
>>> p[2] = 'squirrel'
>>> p
['cat', 'dog', 'squirrel', 'llama']
>>> p[0:2] = ['zebra', 'monkey']
>>> p
['zebra', 'monkey', 'squirrel', 'llama']
```

Lists versus Arrays

- Although lists kind of look like arrays, they are not the same as an array.
  - The elements of a list may be a mixture of variables and objects of different types
  - The elements of an array must be of the same data type.

- Python does have arrays, but we won't be using them.
  - Instead be using arrays from the Numerical Python (NumPy) library.

Functions and Methods for Lists (some also work for tuples)

- len(*ls*) returns the number of items in the list *ls*
- del *ls*[*i*:*j*] deletes items at indices *i* through *j*–1
- *ls*.append(*elem*) adds element *elem* to end of list
- *ls*.extend(*elems*) adds the multiple elements *elems* to the end of the list. Note that *elems* must also be in the form of a list or tuple.

Functions and Methods for Lists (some also work for tuples)

- *ls*.count(*target*) this method returns the number of instances of *target* contained in the list.

- *ls*.index(*target*) this method returns the first index of the list that contains *target*. If optional *i* and *j* are given, it returns first index of occurrence in the range *i* through *j*–1.

- *ls*.insert(*i, elem*) this method inserts *elem* at index *i*

- *ls*.pop(*i*) this method returns element at index *i* and also removes element from the list.

Functions and Methods for Lists (some also work for tuples)

- *ls*.remove(*target*) this method removes first occurrence of target from the list

- *ls*.reverse()this method reverses the list in place

- *ls*.sort()this method sorts the list in place. If keyword reverse = True then it also reverses the results of the sort.

- Note that the reverse() and sort() methods both change (mutate) the actual list. They don't just return a copy.

The range() Function

- The built-in range() function provides a useful means of generating sequences of integers.

```
>>> r = range(-5,8)
>>> r
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7]
```

Caution!

- Notice that the sequence is always one short of the final number in the argument.

- This is true almost everywhere in Python.
  - Ranges and sequences of values do not include the last item in the specified range.

The range() Function (cont.)

- Can use steps, or even go in reverse.

```
>>> r = range(-5,8,3)
>>> r
[-5, -2, 1, 4, 7]
>>> r = range(8, -5, -3)
>>> r
[8, 5, 2, -1, -4]
```

Dictionaries

- A dictionary is a collection of objects that are referenced by a *key* rather than by an index number.

- In other programming languages dictionaries are referred to as *hashes* or *associated arrays*.

Dictionaries

- Dictionaries are defined using curly braces, with the *key*:*value* pairs separated by a colon.
- Elements are accessed by using the key as though it were an index.

```
d = {'first':'John', 'last':'Doe', 'age':34}
>>> d['first']
'John'
>>> d['age']
34
```

Alternate Means of Creating Dictionaries

```
>>> d = dict(first = 'John', last = 'Doe', age = 34)

>>> d = dict([['first','John'], ['last', 'Doe'], ['age', 34]])
```

From a nested list.

Dictionaries are Mutable

```
>>> d
{'age': 34, 'last': 'Doe', 'first': 'John'}
>>> d['age'] = 39
>>> d
{'age': 39, 'last': 'Doe', 'first': 'John'}
```

Functions and Methods for Dictionaries

- len(*d*) returns the number of items in *d*
- del *d*[*k*] removes the item in *d* whose key is *k*.
- *k* in *d* used to see if *d* contains an item with key given by *k*.
  - Returns either True or False.
- *d*.clear() deletes all items in the dictionary

Functions and Methods for Dictionaries

- *d*.copy() makes a copy of the dictionary
- *d*.keys() returns a list of all keys in the dictionary
- *d*.items() returns a list containing tuples of all key-value pairs.
- *d*.values() returns a list of all values in the dictionary

Finding an Object's Type

- The data type of an object can be found by using the type(*obj*) function.

```
>>> a = 4
>>> type(a)
<type 'int'>
>>> b = 4.5
>>> type(b)
<type 'float'>
>>> c = 'Hello'
>>> type(c)
<type 'str'>
>>> d = 4+7j
>>> type(d)
<type 'complex'>
>>> e = (4, 7, 2.3, 'radish')
>>> type(e)
<type 'tuple'>
```

Shallow Copy vs. Deep Copy

- One quirk of Python is that for <u>*compound objects*</u> (e.g., lists, tuples, dictionaries) using the equals sign only creates a *shallow-copy* which uses pointers or references to the original object.

```
>>> a = ['cat', 'dog', 'mouse']
>>> b = a
>>> a,b
(['cat', 'dog', 'mouse'], ['cat', 'dog', 'mouse'])
>>> a[1] = 'frog'
>>> a,b
(['cat', 'frog', 'mouse'], ['cat', 'frog', 'mouse'])
```

Creates a shallow copy

Creating a Deep Copy

- If you truly want an independent copy of a compound object you must use the copy.deepcopy() function.

```
>>> import copy
>>> a = ['cat', 'dog', 'mouse']
>>> b = copy.deepcopy(a)
>>> a,b
(['cat', 'dog', 'mouse'], ['cat', 'dog', 'mouse'])
>>> a[1] = 'frog'
>>> a,b
(['cat', 'frog', 'mouse'], ['cat', 'dog', 'mouse'])
```

Creates a deep copy