Objects, Object Types and Functions

Goals for this Section: "Objects, Types, Functions"

- **Identify** *parts of speech* in code.
- **Determine** the object *type* and *value* of any *variable* / *object*.
- **Describe** an *algorithm* (code block) in English, using specific terms.
- **Compose code** to accomplish basic data calculations.
- **Identify** *exception types* and **debug** simple code issues.

Programs process data.

All programs hold data in *memory registers* and then use the values held there to perform *computations*.

The simplest example is a calculator -- it contains two memory registers -- the **M** memory register and the current calculated value.

Another more visual example is an Excel spreadsheet -- it contains many values stored in cells which can be summed, averaged, etc.

Python stores all of its computational values in abstractions called *objects*.

Object: a data value of a particular *type*

The *object* is Python's abstraction for handling data. We do all computations with objects.

```
var = 100        # assign integer object 100 to variable var

var2 = 100.0     # assign float object 100.0 to variable var2

var3 = 'hello!'  # assign str object 'hello' to variable var3

# NOTE:  'hash mark' comments are ignored by Python.
```

At every point you must be aware of the type and value of every object in your code.

The three object types we'll look at in this unit are **int**, **float** and **str**. They are the "atoms" of Python's data model.

NOTE: variables in these examples will be called **var**, **var2**, **vara**, etc., or **aa**, **bb**, **xx**, etc. These names are arbitrary and determined by the person who created the code. All other names are built-in to Python.

Variable Assignment and Operators

All programs store data as *objects* in *variables* and compute data with *operations*.

```
xx = 10        # assign/initialize integer 10 to variable xx

yy = xx * 2    # compute 10 * 2 and assign integer 20 to variable yy

print yy       # print 20 to screen
```

xx is a *variable* bound to an *integer object* with value **10**
= is an *assignment operator* assigning **10** to **xx**
10 is an *integer object*, value **10**

yy is another *variable* bound to an *integer object* with *value* **20**
***** is a *multiplication operator* computing its *operands* (*integer objects* **10** and **2**)
print is a *statement* that renders its arguments to the screen.

Math Operators with Numbers: +, -, *, /

Math operators behave as you might expect, with one exception: integer division.

```
var = 5
var2 = 10.3

var3 = var + var2    # int plus a float:  15.3, a float

var4 = var3 - 0.3    # float minus a float:  15.0, a float

var5 = var4 / 3      # float divided by an int:  5.0, a float
```

Here is an example of integer division: the result is always an int, which can be confusing:

```
var = 7
var2 = 3

var3 = var / var2    # 2, an int
```

We resolve this by 'converting' one of the operands to a float -- see 'Conversion Functions' coming up.

Reading code: deducing type based on Python's object type rules

Object types are important to Python -- they determine what is possible (and not possible) in the language. Therefore their "behavior" must be thoroughly understood and in some cases memorized to write effective code.

int with int returns an int

```
tt = 5          # assign an integer value to tt
zz = 10         # assign an integer value to zz

qq = tt + zz    # compute 5 plus 10 and assign integer 15 to qq
```

qq is an integer (the sum of two **int** objects)

float with float returns a float

```
aa = 10.0       # assign a float value to aa
bb = 5.0        # assign a float value to bb

cc = aa + bb    # compute 10.0 plus 5.0 and assign float 15.0 to cc
```

cc is a float (the sum of two **float** objects)

int with float returns a float

```
mm = 10         # assign an int value to mm
nn = 10.0       # assign a float value to nn

oo = mm + nn    # compute 10 plus 10.0 and assign float 20.0 to oo
```

oo is a float (the sum of an **int** object and a **float** object)

str plus str return a new, concatenated str

```
kk = 'hello, '  # assign a str value to kk
rr = 'world!'   # assign a str value to rr

mm = kk + rr    # concatenate 'hello, ' and 'world!' to a new str object,

print mm        # 'hello, world!'
```

mm is a **str** (two **strs** concatenated)

Remember, the object types determine the behavior with operators:
- an **int** and an **int** in a math expression produces a new **int**
- a **float** and a **float** in a math expression produces a new **float**
- an **int** and a **float** in a math expression produces a new **float**
- a **str** added to a **str** returns a new **str**

Math Operators with Numbers: exponientation (**)

The exponientation operator (**) raises its left operand to the power of its right operand and returns the result as a float or int.

```
var = 11 ** 2    # "eleven raised to the 2nd power (squared)"
print var        # 121

var = 3 ** 4
print var        # 81
```

Math Operators with Numbers: modulus (%)

The modulus operator (%) shows the remainder that would result from division of two numbers.

```
var = 11 % 2       # "eleven modulo two"
print var          # 1   (11/2 has a remainder of 1)


var2 = 10 % 2      # "ten modulo two"
print var2         # 0   (10/2 divides evenly:  remainder of 0)
```

Because a modulo value of 0 shows that one number divides evenly into the other, we can use it for things like **checking to see if a number is even** or **searching for prime numbers**.

+ Operator with Strings: concatenation

The plus operator (+) with two strings returns a concatenated string.

```
aa = 'Hello, '
bb = 'World!'

cc = aa + bb       # 'Hello, World!'
```

Note that this is the same operator (+) that is used with numbers for summing. Python uses the type to determine behavior.

* Operator with One String and One Integer: string repetition

The "string repetition operator" (*) creates a new string with the operand string repeated the number of times indicated by the other operand:

```
aa = '*'
bb = 5

cc = aa * bb     # '*****'
```

Note that this is the same operator (*) that is used with numbers for multiplication. Python uses the type to determine behavior.

Functions

Built-in functions take *argument(s)* and return *return value(s)*. The arguments are objects *passed* to the function and *returned* from the function (return values).

```
aa = 'hello'            # assign string 'hello' to variable xx

bb = len(aa)            # pass string object aa as an argument to function le
                        # which returns an integer object as a return value.

print bb                # 5  (bb is an integer object)
```

* All functions are *called*: the parentheses after the function name indicate the call.
* All functions take *argument(s)* and return *return value(s)*.
   - The *argument* (or comma-separated list of arguments) is placed in parentheses.
   - The *return value* of the *function call* can be *assigned* to a new *variable*. (It can also be printed or used in an expression.)

Function: len()

The **len()** function takes a string argument and returns an integer -- the length of (number of characters in) the string.

```
varx = 'hello, world!'

vary = len(varx)    # 13
```

Function: round()

The **round()** function takes a float argument and returns another float, rounded to the specified decimal place.

With one argument (a float), **round()** rounds to the nearest whole number value.

```
aa = 5.9

bb = round(aa)      # 6.0
```

With two arguments (a float and an int), **round()** rounds to the nearest decimal place.

```
aa = 5.9583

bb = round(aa, 2)   # 5.96
```

Function: raw_input()

The **raw_input()** function takes a string argument and then *pauses execution*, allowing you (the person running the program) to type characters. It returns a string containing the typed characters.

```
cc = raw_input('enter name:  ')    # program pauses!  Now the user types.

print cc                           # [a string, whatever the user typed]
```

The string prompt is optional, but it's recommended since without it, it can be hard to know why the program paused execution -- you might think it crashed or is hanging.

Built-in Function: exit()

The **exit()** function terminates execution immediately. An optional error message can be passed as a string.

```
aa = raw_input()
if aa == 'q':
    exit(0)                        # indicates a natural termination

if aa == '':                       # user typed nothing and hit [Return]
    exit('error:  input required')  # indicates an error led to terminati
```

Note: the above examples make use of **if**, which we will cover upcoming.

We can also use **exit()** to simply stop program execution in order to debug:

```
aa = '55'
bb = float(aa)
print 'type of bb is', type(bb)
exit()                     # we inserted this to stop the code from contin
                                  # we'll remove it later

cc = bb * 2                       # program continues, but this code will
```

Function: type()

The **type()** function takes any object and returns its type.

```
aa = 5
bb = 5.5
cc = 'hello'

print type(aa)    # <type 'int'>
print type(bb)    # <type 'float'>
print type(cc)    # <type 'str'>
```

We can use this in debugging, since Python cares about type in many of its operations.

Conversion Functions: int(), float() and str()

The **conversion functions** are named after their types -- they take an appropriate value as argument and return an object of that type.

int(): return an int based on a float or int-like string

```
# str -> int
aa = '55'
bb = int(aa)          # 55 (an int)
print type(bb)        # <type 'int'>


# float -> int
var = 5.95
var2 = int(var)       # 5:  the rest is lopped off (not rounded)
```

float(): return a float based on an int or numeric string

```
# int -> float
xx = 5
yy = float(xx)      # 5.0


# str -> float
var = '5.95'
var2 = float(var)   # 5.95 (a float)
```

str(): return a str based on any value

```
var = 5
var2 = 5.5

svar = str(var)     # '5'
svar2 = str(var2)   # '5.5'

print len(svar)     # 1
print len(svar2)    # 3
```

Conversion Challenge #1: treating a string like a number

Numeric data sometimes arrives as strings (e.g. from **raw_input()** or a file). Use **int()** or **float()** to convert to numeric types.

Review: two numeric types added together produce a new number.

```
aa = 5
bb = 5.05

cc = aa + bb       # 10.05, a float
```

Review: two string types 'added' together (concatenated) produce a new string.

```
xx = '5.5'
yy = '5.05'

zz = xx + yy
print zz           # '5.55.05', a str
```

Remember that **raw_input()** produces a string. If the intended input is numeric, we must convert:

```
aa = raw_input('enter number and I will double it:  ')

print type(aa)          # <type 'str'>

num_aa = int(aa)        # int() takes the user's input as an argument
                        # and returns an integer

print num_aa * 2        # prints the user's number doubled
```

You can use **int()** and **float()** to convert strings to numbers.

Conversion Challenge #2: treating an int like a float

Division of two integers results in an integer -- any remainder is discarded. Use **float()** to convert one operand to a float.

Recall that *any math expression* involving two **int** values produces an **int**.

```
var1 = 5
var2 = 5

var3 = var1 + var2            # var3 is 10, an int
```

But what happens when we divide one **int** into another? Does the rule still hold?

```
var1 = 5
var2 = 2

var3 = var1 / var2
print var3                    # var3 is 2, an int
```

It does! Even though we know 5/2 to be 2.5, Python ignores the value and uses the objects' *types* to determine the resulting type.

Remember, an object is a *value* with characteristic *behavior*. This "**int** used with **int** returns an **int**" behavior is consistent and dependable.

The solution is to convert one of the operands to **float** so that the result is a **float**.

```
var1 = 5
var2 = 2

var3 = var1 / float(var2)
print var3                   # var3 is 2.5, a float
```

You will need to employ the solutions for both of these conversion challenges in the homework.

Exceptions

An **Exception** is *raised* when you ask Python to do something that it can't understand, or that breaks one of its rules.

```
var1 = 'Hello, '
var2 = 'World!
var3 = var1 + var2
print var3
```

the above code raises the following error:

```
  File "./test.py", line 4
    var2 = 'World!
                 ^
SyntaxError: EOL while scanning string literal
```

This exception indicates that a close quote is missing.

Some students develop the habit of simply "trying something else" when they get an error. It's extremely important not to skip reading your exception message, as it is trying to tell you what's wrong with the code.

More importantly, understanding each exception message will help you understand how to "think like Python" and avoid the problem in future.

Exceptions: SyntaxError

A **SyntaxError** is rasied when code syntax is incorrect.

```
var1 = 'Joe
var2 = 'Below'
var3 = var1 + var2
print var3
```

above code raises this error:

```
  File "./test.py", line 3
    var1 = 'Joe
               ^
SyntaxError: EOL while scanning string literal
```

"EOL" means End of Line; "string literal" refers to the intended string 'Joe'. Python is telling us that it got to the end of the line before encountering the close quote that was started with **'Joe**

When you get a SyntaxError, look closely at the line to make sure there are no missing quotes, commas, parentheses, operators, etc.

Exceptions: NameError

A **NameError** is raised when Python sees a variable name that hasn't been created.

```
var1 = 'Joe'
var2 = Below
var3 = var1 + var2
```

above code raises this error:

```
Traceback (most recent call last):
  File "./test.py", line 4, in
    var2 = Below
NameError: name 'Below' is not defined
```

In line 2 we attempted to create a string **Below** but we forgot to put quotes around it. Python assumed we meant a *variable* named **Below** and complained that we hadn't defined one.

Exceptions: TypeError

A **TypeError** is raised when an object of the wrong type is used in a statement.

```
var1 = 5
var2 = '5'
var3 = var1 + var2
```

above code raises this error:

```
Traceback (most recent call last):
  File "./test.py", line 5, in
    var3 = var1 + var2
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python is telling us that you can't add a string to an integer. If your purpose was to sum these numbers, you'd want to convert the **str** to a **int** using the **int()** function.

```
var1 = 'Joe'
var2 = 55
var3 = len(var1)     # assign int 3 (len() of 'Joe') to var3

var4 = len(var2)
```

above code raises this error:

```
Traceback (most recent call last):
  File "./test.py", line 7, in
    var4 = len(var2)
TypeError: object of type 'int' has no len()
```

Python is telling us that you can't get the length of an **int**. If your purpose was to find out how many digits were in **55**, you'd want to convert the **int** to a **str** using the **str** function.

Reading code: identifying the 'parts of speech'

The prerequisite for writing code is the ability to read code. Identifying the "parts of speech" is the first step.

```
var1 = 'Joe'          # var1:  variable
                      # 'Joe':  object

var2 = len(var1)      # var2:  variable, also return value
                      # len():  function
                      # var1:  variable, also argument

var3 = var2 * 2       # var3:  variable
                      # var2:  variable
                      # 2:  object

print var3            # print:  statement
                      # var3:  variable
```

Reflect on the above code and note the definitions and terminology for each element. It is required that you be able identify each element of code.

variable	a name bound to an object
object	a data entity of a particular type
function	a named routine called with an argument; returns a return value
operator	an entity that computes or processes its object operands
statement	similar to a function, it processes objects

Scroll down to test yourself on the above.

In the below code:

```
var1 = 'Joe'
var2 = len(var1)
var3 = var2 * 2
print var3
```

<u>Identify the following</u>:

```
•   var1 (2 IDs)
•   'Joe'
•   var2 (2 IDs)
•   *
•   2
•   len()
•   var3
•   print
```

Reading code: tracing program execution

Next, reading code means knowing the *type* and *value* of *every object* in our code.

```
var1 = 'Joe'           # str 'Joe' assigned to var1
var2 = 2               # int 2 assigned to var2
var3 = 'Below'         # str 'Below' assigned to var3
var4 = len(var1)       # int 3 (len() of 'Joe') assigned to var4
var5 = var1 + var3     # str 'JoeBelow' (two strs concatenated) assigned
var6 = len(var5)       # int 8 (len() of 'JoeBelow') assigned to var6
var7 = var6 * var2     # int 16 (8 * 2) assigned to var7

print var7             # 16
```

Always insist on knowing the identity (*type* and *value*) of every object in your code.

Writing code: comments

Use hash marks to comment individual lines; use "triple quotes" to comment multiple lines.

```
# here is a comment that will be ignored
var1 = 'Joe'
var2 = 2
var3 = 'Below'
# var4 = len(var1)   (this line was commented because we didn't need it)
var5 = var1 + var3

"""                   # these 2 lines are commented (ignored)
var6 = len(var5)
var7 = var6 * var2
"""
print var5
```

Writing code: debugging Technique

Use print statements to determine the *type* and *value* of every object in your code.

Suppose you're working on the below code. You were expecting 16 (as it was in the prior example) but instead you're getting 24 (as shown below). How do you elucidate?

```
var1 = 'Joe'
var2 = 2
var3 = 'Below'
var4 = len(var1)
var5 = var1 + var3
var6 = len(var5)
var7 = var6 * var4

print var7             # 24
```

You can read the code and keep track of the variable types and values yourself.

This can work, but keep in mind that if you're incorrect about a values resulting from any step, you will be incorrect about values that depend on that value -- as you have been able to see, these values are handed off and passed along from variable to variable.

You can insert print statements.
This is probably better in most situations. The upside is that at any point you can see what your code is doing -- what is the value and/or type of any object in your code. (The only downside is that if your printed output becomes confusing it may confuse what is happening in your code.)

Sidebar: float precision and the Decimal object (1/3)

Because they store numbers in binary form, all computers are incapable of absolute float precision -- in the same way that decimal numbers cannot precisely represent 1/3 (0.33333333... only an infinite number could reach full precision!)

Thus, when looking at some floating-point operations we may see strange results. (Below operations are through the Python interpreter.)

```
>>> 0.1 + 0.2
0.30000000000000004          # should be 0.3?

>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17        # should be 0.0?

>>> round(2.675, 2)
2.67                         # round to 2 places - should be 2.68?
```

Sidebar: float precision and the Decimal object (2/3)

What's sometimes confusing about this 'feature' is that Python doesn't always show us the true (i.e., imprecise) machine representation of a fraction:

```
>>> 0.1 + 0.2
0.30000000000000004

>>> print 0.1 + 0.2
0.3

>>> 0.3
0.3
```

The reason for this is that in many cases Python shows us a *print representation* of the value rather than the actual value. This is because for the most part, the imprecision will not get in our way - rounding is usually what we want.

Sidebar: float precision and the Decimal object (3/3)

However, if we want to be sure that we are working with precise fractions, we can use the **decimal** library and its **Decimal** objects:

```
from decimal import Decimal          # more on module imports later

float1 = Decimal('0.1')     # new Decimal object value 0.1
float2 = Decimal('0.2')     # new Decimal object value 0.2
float3 = float1 + float2    # now float is Decimal object value 0.3
```

Although these are not **float** objects, **Decimal** objects can be used with other numbers and can be converted back to **floats** or **ints**.

Glossary

- argument: a value that is passed to a function for processing
- assign: bind an object to a name
- exception: Python's error condition, raised when there is a problem
- function: a "routine" that performs
- initialize: establish (or re-establish) a new variable through assignment
- object: a value of a particular type (int, float, str, etc.)
- operand: the values on either side of an *operator*
- operator: an entity that processes its *operands*
- raise: when Python signals an error
- return value: a value that is returned from a function after processing is done
- statement: a line or portion of code that accomplishes something
- type: a classification of objects. Every object has at ype
- variable: an object assigned ("bound") to a name