2.2 - Python Basics

Our goal here is to move through an ultra-condensed version of "Computer Science 101" -- to be good data scientists we need to have a good familiarity working with code.

Put another way, no one wants to live in a house built on a weak foundation!

Outline:

1. Arithmetic
2. Objects & Object Methods
3. Control Flow
4. Examples
5. Functions
6. String formatting

1. Let's begin with simple arithmetic.

First, we calculate some basic expressions:

In [6]:

```
# If we want Jupyter to report the output of some calculation, wrap it
# in print()
print(1 + 3 - 4 * 5 / 6)     # +, -, *, and / are used for standard oper

print(1 +(3 - 4)* 5 / 6)     # making grouping clear with parentheses is

print(3*2**2)                # exponents are denoted with **, not ^

# There are three types of division (!) in Python
# (and pretty much every language)
print(6/4)                   # usual floating point division; this gives
print(6//4)                  # Integer or "floor" division; round down
                             # "regular" division to the nearest integer
print(6%4)                   # Modulus division; what is the remainder w
                             # doing integer division?

```

```
0.6666666666666665
0.16666666666666663
12
1.5
1
2
```

The order of operations follows the usual `PEMDAS`, except...

It is better to say something like `P E MD AS` because...

- Multiplication and Division are evaluated "with equal precedence", left to right

- Multiplication and Division are evaluated "with equal precedence", left to right
- Addition and Subtraction are evaluated "with equal precedence", left to right

Note also that exponentiation is applied "right to left", as is standard practice.

For example, `2**3**4 = 2**(3**4)`

Try predicting the output of the following *before* you check:

- `4//3*4`
- `2**8/2*2**4/2`

In [7]:
```
print(4//3*4)
print(2**8/2*2**4/2)
```

```
4
1024.0
```

Explanations for the above:

- Multiplication and division (all forms) take equal precedence, so go left-to-right: `4//3 = 1`, then `1*4=4`.
- Exponents take precedence, so we have

```
  2**8/2*2**4/2
= (2**8)/2*(2**4)/2   # exponents first
= 256/2*16/2          # now just left to right
= 128*16/2
= 1024
```

2. Evaluating "simple" mathematical expressions like this isn't very powerful or interesting. We obtain greater flexibility with *objects*.

In the simplest case, this lets us record some computations by "giving it a name":

In [8]:

```
# A good way to read "x = <whatever>" is "x becomes"
# or "x is assigned to be" whatever follows the "="

x = 3%6   # x becomes the result of 3%6
print(x)
x = 3*x   # x becomes 3 multiplied by the current value of x
print(x)
x *= 3    # this is "augmented assignment" and is the same as x = 3*x
          # you can also do /=, +=, -=, etc.
print(x)

```

```
3
9
27
```

There are many types of objects. We can check what we have with the `type()` command.

Try this:

In [9]:

```
x = 3
print(type(x))  # notice this is evaluated "inside out": do type(x),
                # then print that

y = 3.0
print(type(y))

# This demonstrates that Python differentiates between integers
# (whole numbers) and floating point numbers (with something
# after the decimal, even if it is 0)

```

```
<class 'int'>
<class 'float'>
```

The usefulness of storing numbers is fairly obvious; it allows you to carry out multiple computations in sequence, carrying forward the results each time.

What are some other common types of objects in Python?

- **Dataframes** from the pandas module. We saw last time (notes 2.01) that a dataframe essentially contains an entire table, like an Excel spreadsheet or data pulled from a "comma seperated values" (.csv) document.

- **Strings**, which is just regular text, defined like so: `x = "A string!"` This is useful when we want our scripts to provide meaningful output to a reader... or if we're manipulating text in the first place!

- **Lists**, defined with square brackets like so: `x = [1,2,3]` . We will use these frequently; for example, making a scatter plot of two columns of data against one another is most simply done with two lists of the values (e.g. `x=[1,2,3]` and `y=[5,6,7]`).

- **Booleans**, which are logical values `True` and `False` , defined with e.g. `x = True` .

We'll ignore dataframes for now, but let's play a bit with the other three. Every type of object has built in **methods**, which are essentially standard routines we can access directly with **object dot** notation. Let's demonstrate with a silly example.

In [10]:

```python
# Create a string object
x = "a silly example"
print(x)

# Spit out just one character
# The index starts at 0 and increases from there
print(x[0])

# Or we can do a "slice" with x[start:end]
print(x[2:6])

# Use a method to make it all capitalized
print(x.upper())
print(x)         # Notice that doing x.upper() doesn't change x
x = x.upper()
print(x)         # But explicitly making a reassignment does!
```

```
a silly example
a
sill
A SILLY EXAMPLE
a silly example
A SILLY EXAMPLE
```

Let's play for a minute. Start with `x = " Python is awesome "` and experiment with the following string methods:

1. `x.capitalize()`
2. `x.lower()`
3. `x.isupper()`
4. `x.islower()`
5. `x.isnumeric()`
6. `x.isalpha()`
7. `x.isalnum()`
8. `x.count("a")` (for all these, "a" just means any string)

9. `x.find("a")`
10. `x.split()` or `x.split("a")`
11. `x.startswith("a")`
12. `x.endswith("a")`
13. `x.replace("a","b")`
14. `x.lstrip()`
15. `x.rstrip()`

In [11]:
```
x = " Python is awesome "

# change the below line...
y = x.split("a")

print(y)
```

```
[' Python is ', 'wesome ']
```

Generating lists

As we turn to lists, let's think first see how to create them.

In [12]:
```
# The first method we've already seen: be explicit
x = [1,2,3]

# The "slice" notation is the same as for string
print(x[2])

# Another method is to use a generator:
# we specify where we start and stop
x = list(range(0,5))
print(x)

# Notice that the last value is actually 4, not 5
# This is to ensure there are 5-0=5 entries in the list
```

```
3
[0, 1, 2, 3, 4]
```

List methods

OK, let's play with list methods. A few examples:

In [13]:

```
x = []        # Create an empty list

x.append(1)   # append an entry to the end of the list
x.append(2)
print(x)

x = [10,3,12,7,3]    # A different list
x.sort()             # Change it to ascending order
print(x)

x.reverse()          # Flip the order (here, descending)
print(x)

print(x.count(3))    # how many entries = 3 do we have?

```

```
[1, 2]
[3, 3, 7, 10, 12]
[12, 10, 7, 3, 3]
2
```

Your turn! See what the following list methods do:

1. `x.index(12)`
2. `x.insert(100,2)`
3. `y = x.pop()`
4. `x.remove(10)`

In addition, there are some *built in* functions in Python that are useful when applied to lists (these don't use the **object dot** notation because they're not specific to lists):

1. `min(x)`
2. `max(x)`
3. `len(x)`

In [27]:

```
# Let's play with the above suggestions...

# Make the list
x = [10,3,12,7,3]

# play with the below line
x.insert(1,2)
#y = x.pop()

print(x)
#print(y)
```

```
[10, 2, 3, 12, 7, 3]
```

Booleans

As a final example (for now!) of objects in Python, let's take a look at Booleans. There are just two; `True` and `False` ; Python generates these when you present it with a logical statement. For example:

In [15]:
```
# We read this as "x is assigned to be ...
# the result of the statement '5 is greater than 4',
# which is True.

x = 5>4
print(x)

# Python can handle different types
# as long as there is a logical way to do so
print(200 == 200.0)

# This one is a bit more subtle
# compare the position in the alphabet of the first
# characters (P,J); if they're the same check the
# second, and so on, until a mismatch is found
print('Python' < 'Java')
```

```
True
True
False
```

Booleans are particularly useful as we move on to...

3. For anything but the simplest scripts we want to do different things in different situations. This is so-called "*control flow*"

In the simplest case, Python runs the first line of a script, then the second, all the way to the end.

What if we want to run some code **conditionally**? One example of so-called *control flow* is when we want to do something like:

"if **something is True**, do **this**; otherwise, **do that**"

This is implemented with an "if/else" or "if/elif/else" statement:

In [16]:

```python
x = 10

if x > 100:
    # x > 100 evaluates to either True or False (a Boolean),
    # if it evals to True, we do the line indented below
    print("x is bigger than 100.")
else:
    # if the statement at the top is False, we do the below
    # line instead
    print("x is less than or equal to 100.")

```

```
x is less than or equal to 100.
```

Notice the syntax:

```
if <something that evalues to True or False>:
    #indented code that fires if the above statement evaluates to True
else:
    #a catch-all that fires if the above statement evaluates to False
```

We can expand this structure with multiple checks, like so:

```python
if x > 100:
    print("x is bigger than 100.")
elif x > 50:
    print("x is less than or equal to 100, but bigger than 50.")
else:
    print("x is less than or equal to 50.")
```

We sometimes also want to give an instruction along the lines of:

"Do **something** to every item in **some container.**"

(Formally, a container is an *iterable*, an object that can spit out its members one at a time.)

or

"Do **something** a certain number of times"

This is implemented with a "for loop":

In [17]:
```
x = [1,2,3,4]

# Simple example: just output the entries
for i in x:
    print(i)

# maybe we'd like to compute the sum of the squares
# of the entries in the list
value = 0
for i in x:
    value += i**2

print(value)

# Or if we don't want to use a pre-defined list:
for i in range(5):
    print(i)
```

```
1
2
3
4
30
0
1
2
3
4
```

Finally, sometimes we want to continue a calculation multiple times, like in a for loop, but we don't know ahead of time how many calculations we might like to do. We can instead say something like

"While **some condition** is True, do **something**

This is implemented with a "while" loop:

In [18]:

```python
# Maybe we want to add up the integers squared, and
# keep going until the sum is one million. What is
# the last integer we add?

counter = 0
value = 0

while value < 1E6:
    # we do whatever is indented below until value < 1E6
    # becomes False.
    counter += 1
    value += counter**2


# once we're done, we print out both objects
print(counter)
print(value)

# We see that when we added 144**2, we exceed one million
# So the last integer we'd add to -not- exceed one million
# is 143.
```

```
144
1005720
```

4. Examples

Phew! Let's stop and apply some of what we've done. Consider the following examples...

Example 1

You can find the maximum value in a list with `max(x)` . For practice, let's write our own approach:

- Snag the first item and say "as far as I know, this is the biggest"; call it m.
- Check the second entry. If it is larger than m, update m to be this new value.
- Repeat the above step as you consider every remaining item in the list.

Check this with the following list: `x = [5, 8, 120, 4]`

Example 2

Use a **for loop** to calculate the sum

$$v = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$$

where you stop the sum after including 20 terms in the series.

Example 3

The series from example 2 converges to $v = 1$ as you include more and more terms. How many terms do you need to include to come within 0.001% of this value?

To calculate percent error P, use the formula

$$P = \frac{|\text{actual} - \text{theoretical}|}{\text{theoretical}} \times 100$$

In [19]:

```
# Solutions for the examples (there are multiple ways to do these)

# Example 1
x = [5,8,120,4]

m = x[0]

# Look at every entry in x
for i in x:
    # Check if the one under consideration is bigger than
    # the current "biggest so far"
    if i > m:
        # If it is, update
        m = i

# print
print(m)
```

```
120
```

In [20]:

```
# Example 2

v = 0

# i will be 1, 2, 3, ..., 20
# we want to add up terms like 1/(2**i)

for i in range(1,21):
    v += 1/(2**i)

print(v)

```

```
0.9999990463256836
```

In [21]:

```
# Example 3

i = 1         # Use this to "walk across" greater terms
v = 0         # Running sum
tol = 0.001   # How close do we want to get?
P = 1         # Our percent error. Create it as something big
              # to start loop

while P > tol:
    v += 1/(2**i)
    i += 1

    P = 100*abs(v-1)/1.

# We increased i inside the loop after updating v, so "de-count"
print(i-1)

```

```
17
```

5. Functions

In more complex scripts, we might want to run a certain bit of code multiple times. Defining a function "puts the code in a drawer" and allows us to pull it out whenever we need to.

This makes code tidier (less repetition) and easier to update/debug.

Using a function conceptually involves three steps:

1. Feed in some information (i.e, provide the *arguments* of the function)
2. Do something with the arguments (e.g., add them)
3. Provide some information back to the main script (e.g., `return` the sum)

A simple example will hopefully make this clear:

In [22]:
```
# Define the function

def f(x,y):
    # x and y are the arguments
    out = x + y
    # `return` says "provide what follows to the main script"
    return out

# If we only define a function, the script doesn't do anything;
# to use it we have to "call" it:
v = f(4,5)

# Here, v becomes whatever is returned from the function
# (here, what we called `out`)
print(v)
```

```
9
```

Function Example

The binomial approximation says that if $x \ll 1$,

$$(1 + x)^n \approx 1 + nx$$

You can check how good the approximation is by computing the percent difference (the left side being exact and the right being approximate).

Write a function that takes x and n as its arguments and returns the percent difference. Run it for the following:

1. $n = 2$, $x = 0.01$
2. $n = 2$, $x = 0.001$
3. $n = 2$, $x = 0.0001$
4. $n = 4$, $x = 0.01$
5. $n = 4$, $x = 0.001$
6. $n = 4$, $x = 0.0001$

In [23]:

```
def check(n,x):
    exact = (1+x)**n
    approx = 1+n*x

    P = 100*abs(approx-exact)/exact

    return P

print(check(2,0.01))
print(check(2,0.001))
print(check(2,0.0001))

print(check(4,0.01))
print(check(4,0.001))
print(check(4,0.0001))
```

```
0.009802960494068 13
9.980029957012908e-05
9.99800023919745e-07
0.05804417378710545
0.0005980044915624026
5.998000439015783e-06
```

6. String Formatting

Thus far, when we have reported our results we have used a simple print statement, like `print(x)` . For complicated programs we would like to give more nicely formatted output. We can do thing with "string formatting".

The idea is fairly straightforward: use "curly braces" to set up a placeholder in a string, then specify

In [24]:

```
# Simple example
print("My favorite color is {}.".format("blue"))

# You can have more than one placeholder:
print("My favorite color is {} and my favorite food is {}.".format("blu

# You can specify which entry goes where:
print("My favorite color is {0}, {0}, {0}! Also I like to eat {1}.".for

```

```
My favorite color is blue.
My favorite color is blue and my favorite food is PEZ.
My favorite color is blue, blue, blue! Also I like to eat PEZ.
```

There are many ways to customize the way the information is formatted. Try the following and see if you can decipher what is going on:

In [25]:

```
# Make some long decimal
x = 2/11
print(x)

print('The result is {0:1.2f}'.format(x))
print('The result is {0:10.2f}'.format(x))
print('The result is {0:1.3e}'.format(x))
```

```
0.18181818181818182
The result is 0.18
The result is       0.18
The result is 1.818e-01
```

The information after the colon is in the format `n1.n2` and one final character where:

- `n1` specifies the minimum width of the entry (useful to align multiple outputs to the console)
- `n2` specifies how many digits after the decimal to include
- the final character, here either `f` or `e` , specifies "floating point" or "exponential" (scientific) notation.

Example

Go back to the previous example for the binomial approximation. "Prettify" the output.

In [26]:

```
# One approach...

def check(n,x):
    exact = (1+x)**n
    approx = 1+n*x

    P = 100*abs(approx-exact)/exact

    return P

out = check(2,0.01)
print("For n = {0}, x = {1:6}, the percent error is {2:1.3e}".format(2,
                                                                      
out = check(2,0.001)
print("For n = {0}, x = {1:6}, the percent error is {2:1.3e}".format(2,

out = check(2,0.0001)
print("For n = {0}, x = {1:6}, the percent error is {2:1.3e}".format(2,

out = check(4,0.01)
print("For n = {0}, x = {1:6}, the percent error is {2:1.3e}".format(4,

out = check(4,0.001)
print("For n = {0}, x = {1:6}, the percent error is {2:1.3e}".format(4,

out = check(4,0.0001)
print("For n = {0}, x = {1:6}, the percent error is {2:1.3e}".format(4,
```

```
For n = 2, x =   0.01, the percent error is 9.803e-03
For n = 2, x =  0.001, the percent error is 9.980e-05
For n = 2, x = 0.0001, the percent error is 9.998e-07
For n = 4, x =   0.01, the percent error is 5.804e-02
For n = 4, x =  0.001, the percent error is 5.980e-04
For n = 4, x = 0.0001, the percent error is 5.998e-06
```