0b_intro_to_python_for_psycholinguists

March 21, 2021

Open in Colab

1 Intro to Python for psycholinguists

Sources

- This notebook is closely based on a notebook by Todd M. Gureckis introducing python for psychology undergrads.
- The section on for loops was developed by Lisa Tagliaferri.

1.1 Introduction

This notebooks is by no means a proper, systematic introduction to programming with Python. It just introduces the basics needed to do the kind of cognitive modeling we'll be doing in the rest of this course.

1.2 What is Python?

Python is a popular general-purpose programming language that is easy to learn, flexible, and free to use. Python is extensively used in both academia and industry. One reason is that Python has a very strong set of libraries that enable you to use it for all kinds of tasks.

This notebook is divided into different subsections, each reviewing a basic feature of Python3:

- Section 1.12

1.3 Comments

Comments in Python start with the hash character, #, and extend to the end of the line.

- A comment may appear at the start of a line or following whitespace or code
- but not within a string; a hash character within a string is just a hash character

Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

[1]:
```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
print(text)
```

```
# This is not a comment because it's inside quotes.
```

1.4 Calling functions

A **function** in Python is a piece of code, often made up of several instructions, which runs when it is referenced or "called".

- Functions are also called **methods** or **procedures**.

Python provides many default functions (like `print()` above), but also gives you freedom to create your own custom functions; we return to this later.

Functions have a couple of key elements:

- A **name** which is how you specify the function you want to use
- The name is followed by a open and close parentheses `()`
- optionally, a function can include one or more arguments or parameters which are placed inside the parentheses

We already say one function, which you will use a lot, called `print()`. The `print()` function lets you print out the value of whatever you provide as arguments or parameters. For example:

[2]:
```
a = 1
print(a)
```

```
1
```

The code cell above defines a variable `a` and then prints the value of a.

- Notice how in Jupyter the color of the word print is special (usually green).

The `print()` function is itself a set of lower-level commands that determine how to print things out.

To learn more about the `print` function, you can type it in a code cell followed by a question mark, or equivalently, you can call the `help` function on it.

[3]:
```
# print?
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Here's another example built-in function called `abs()` that computes the absolute value of a number:

[4]:
```
abs(2)
```

[4]: 2

[5]:
```
abs(-1)
```

[5]: 1

Again note the syntax: the name of the function followed by the arguments within round brackets `()`.

It is important that the round brackets/parentheses are matched. If you forget closing parentheses, you'll get an error:

[6]:
```
abs(1
```

```
      File "<ipython-input-6-bc02591c6df4>", line 1
    abs(1
         ^
SyntaxError: unexpected EOF while parsing
```

[7]:
```
abs(-1(
```

```
      File "<ipython-input-7-bb8c754202f6>", line 1
    abs(-1(
           ^
```

```
SyntaxError: unexpected EOF while parsing
```

1.5 Using Python as a Calculator

The Python interpreter can act as a simple calculator: type an expression and the interpreter outputs its value.

Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` work just like in most other programming languages; parentheses (`()`) can be used for grouping and order or precedence. For example:

```
[8]: 2 + 2
```

```
[8]: 4
```

```
[9]: 50 - 5 * 6
```

```
[9]: 20
```

You often have to use parentheses to enforce the order of operations.

```
[10]: (50 - 5 * 6) / 4
```

```
[10]: 5.0
```

PLEASE always put blank spaces around the operators and after commas (you'll see what I mean later). In general, **PLEASE** use the coding style in these notebooks rather than cramming everything in as short a line as possible.

If your code does not follow the same style as these notebooks and is hard to read, **I won't read it and you will lose all the points**.

- Reading badly styled code, even if the code works, feels like reading a text riddled with typos. I won't do it, and I'll just take off all the points for that part of the homework / final project.

In most computer programming language, there are multiple "types" of numbers. This is because the internals of the computer deal with numbers differently depending on whether or not they have decimals.

The two types of numbers are:

- Integers (`int`; numbers without decimals)
- Floating-point numbers (`float`; numbers with decimals).

You can check the type of a number using the `type()` function.

```
[11]: type(2)
```

```
[11]: int
```

```
[12]: type(2.0)
```

```
[12]: float
```

```
[13]: type(2.)
```

```
[13]: float
```

```
[14]: type(1.234)
```

```
[14]: float
```

Where it can sometimes get tricky is if you convert between types. For instance dividing two ints always results in a float:

```
[15]: 8 / 5  # Division always returns a floating point number.
```

```
[15]: 1.6
```

```
[16]: type(8), type(5), type(8 / 5)
```

```
[16]: (int, int, float)
```

```
[17]: type(8 / 4)
```

```
[17]: float
```

To do what is known as a floor division and get an integer result (discarding any fractional result), you can use the // operator; to calculate the remainder, you can use % (modulus):

```
[18]: 17 / 3  # Classic division returns a float.
```

```
[18]: 5.666666666666667
```

```
[19]: 17 // 3  # Floor division discards the fractional part.
```

```
[19]: 5
```

```
[20]: 17 % 3  # The % operator returns the remainder of the division.
```

```
[20]: 2
```

This last operation (%) is pretty common or useful. For example, you can use it to cycle through a list of numbers:

```
[21]: numbers = [0,1,2,3,4,5,6,7,8,9,10]
      for i in numbers:
          if i%3 == 0:
```

```
        print(i)
```

```
0
3
6
9
```

We'll talk more about `for` loops below.

The`**` operator can be used to calculate powers:

[22]:
```
5 ** 2   # 5 squared
```

[22]:
```
25
```

[23]:
```
2 ** 7   # 2 to the power of 7
```

[23]:
```
128
```

Python supports other types of numbers in addition to `int` and `float`.

1.6 Variables

One of the most important concepts in programming, and one feature that makes it really useful is the ability to create **variables** to refer to numbers, or any other objects.

Variables are named entities that refer to certain types of data inside the programming language. We can assign values to a variable in order to save a result or use it later.

You can think of variables as buckets, or slots in which you place their values. The variable name is the name bucket/slot, and if you want to access what's inside, you do it by invoking the name of the bucket/slot. - We write the name of the variable on the outside of the bucket and put something in the bucket using assignment.

Let's look at an example. We can create a variable named `width` and one named `height`. The equal sign (=) assigns a value to a variable:

[24]:
```
width = 20
height = 5 * 90
print(width, height)
```

```
20 450
```

[25]:
```
area = width * height
print("area = ", area)
```

```
area =  9000
```

In Python, the first time we assign something to a variable, the variable is created.

- by executing `width = 20`, we created in the memory of the computer a variable named `width` that contains the `int` value 20.

- similarly, we created a variable called `height` which contains the value `5*90`
- then, we created a variable `area` which is the multiplication of `width` and `height`

One nice feature of this naming style is that the code is readable: we understand that this calculation computes the area of a rectangle and which dimensions refer to which parts of the rectangle.

Python syntax also helps a lot with code readability.

We can make up any name for variable, but there are a few simple rules. The rules for the names of variables in Python are: - A variable name must start with a letter or the underscore character (e.g., `_width`) - A variable name cannot start with a number - A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _) - Variable names are case-sensitive (`age`, `Age`, and `AGE` are three different variables)

Accessing the value of an undefined variable will cause an error. For instance we have not yet defined `n`, so asking Jupyter to output its value here will not work:

[26]:
```
n  # Try to access an undefined variable.
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-26-2d383632fd8e> in <module>
    ----> 1 n  # Try to access an undefined variable.


        NameError: name 'n' is not defined
```

We can get a list of all the current named variables in our Jupyter kernel using this `%whos` command, which is a special feature of Jupyter and not part of Python core language:

[27]:
```
%whos
```

```
Variable       Type        Data/Info
-------------------------------------
a              int         1
area           int         9000
height         int         450
i              int         10
json           module      <module 'json' from
'/usr<...>hon3.8/json/__init__.py'>
numbers        list        n=11
spam           int         1
text           str         # This is not a comment b<...>cause it's inside
quotes.
width          int         20
yapf_reformat  function    <function yapf_reformat at 0x7fb2fc04a160>
```

You can see a number of variables here, including the variables `area`, `height`, and `width`.

Variables can hold all types of information, not just single number.

Variables can also be used temporarily to move things around. For instance, let's define variables x and y.

[28]:
```
x = 2
y = 7
```

We would like to swap the values, so that x has the value of y and y has the value of x. To do this, we need to stay organized because if we just assign the value of y directly to x, it will overwrite it. So instead we will create a third "temporary" variable to swap them:

[29]:
```
tmp = x
x = y
y = tmp
print("x = ", x)
print("y = ", y)
```

```
x =  7
y =  2
```

To save space in Python, you can define multiple variables at once on the same line:

[30]:
```
width, height = 10, 20
print("width = ", width)
print("height = ", height)
```

```
width =  10
height =  20
```

This type of compact notation can even be used to more efficiently swap variables:

[31]:
```
x, y = 1, 2
x, y = y, x
print("x = ", x)
print("y = ", y)
```

```
x =  2
y =  1
```

1.7 The basics of working with strings

Besides numbers, Python can also manipulate strings. Strings are small pieces of text that can be manipulated in Python. Strings can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result. Use \ to escape quotes, that is, to use a quote within the string itself:

[32]:
```
'spam eggs'  # Single quotes.
```

[32]:
```
'spam eggs'
```

```
[33]: 'doesn\'t'  # Use \' to escape the single quote...
```

```
[33]: "doesn't"
```

```
[34]: "doesn't"  # ...or use double quotes instead.
```

```
[34]: "doesn't"
```

In the interactive interpreter and Jupyter notebooks, the output string is enclosed in quotes and special characters are escaped with backslashes.

- what does it mean to 'escape with backslashes'?

Although this output sometimes looks different from the input (the enclosing quotes could change), the two strings are equivalent:

- the string is enclosed in double quotes if the string contains a single quote and no double quotes
- otherwise, it's enclosed in single quotes.

The `print()` function produces a more readable output by omitting the enclosing quotes and by printing escaped and special characters:

```
[35]: '"Isn\'t," she said.'
```

```
[35]: '"Isn\'t," she said.'
```

```
[36]: print('"Isn\'t," she said.')
```

```
"Isn't," she said.
```

```
[37]: s = 'First line.\nSecond line.'  # \n means newline.
      s  # Without print(), \n is included in the output.
```

```
[37]: 'First line.\nSecond line.'
```

```
[38]: print(s)  # With print(), \n produces a new line.
```

```
First line.
Second line.
```

Strings can span multiple lines and are delineated by triple-quotes: `"""..."""` or `'''...'''`.

End of lines are automatically included in the string, but it's possible to prevent this by adding a \ at the end of the line.

- for example, without a \, the following example includes an extra line at the beginning of the output:

```
[39]: print("""
      Usage: thingy [OPTIONS]
```

```
     -h                         Display this usage message
     -H hostname                Hostname to connect to
""")
```

```
Usage: thingy [OPTIONS]
     -h                         Display this usage message
     -H hostname                Hostname to connect to
```

Compare with:

[40]:
```
print("""\
Usage: thingy [OPTIONS]
     -h                         Display this usage message
     -H hostname                Hostname to connect to
""")
```

```
Usage: thingy [OPTIONS]
     -h                         Display this usage message
     -H hostname                Hostname to connect to
```

Strings can be *concatenated* (glued together) with the + operator, and repeated with *:

[41]:
```
# 3 times 'un', followed by 'ium'
3 * 'un' + 'ium'
```

[41]:
```
'unununium'
```

To concatenate string-valued variables, or a variable and a string, use +:

[42]:
```
prefix = 'Py'
prefix + 'thon'
```

[42]:
```
'Python'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

[43]:
```
word = 'Python'
word[0]  # Character in position 0.
```

[43]:
```
'P'
```

[44]:
```
word[5]  # Character in position 5.
```

[44]:
```
'n'
```

Indices may also be negative numbers, in which case we start counting from the end of the string.

- because -0 is the same as 0, negative indices start from -1

```
[45]: word[-1]  # Last character.
```

```
[45]: 'n'
```

```
[46]: word[-2]  # Second-last character.
```

```
[46]: 'o'
```

In addition to indexing, which extracts individual characters, Python also supports *slicing*, which extracts a substring.

To slice, we indicate a *range* in the format `start:end`, where the start position is included but the end position is excluded:

```
[47]: word[0:2]  # Characters from position 0 (included) to 2 (excluded).
```

```
[47]: 'Py'
```

If you omit either position, the default start position is 0 and the default end is the length of the string:

```
[48]: word[:2]   # Character from the beginning to position 2 (excluded).
```

```
[48]: 'Py'
```

```
[49]: word[4:]  # Characters from position 4 (included) to the end.
```

```
[49]: 'on'
```

```
[50]: word[-2:] # Characters from the second-last (included) to the end.
```

```
[50]: 'on'
```

This means that `s[:i] + s[i:]` is always equal to `s`:

```
[51]: word[:2] + word[2:]
```

```
[51]: 'Python'
```

```
[52]: word[:4] + word[4:]
```

```
[52]: 'Python'
```

One way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*. For example:

- the first row of numbers gives the position of the indices 0…6 in the string

- the second row gives the corresponding negative indices

The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large results in an error:

[53]:
```
word[42]   # The word only has 6 characters.
```

```
        ---------------------------------------------------------------------------

        IndexError                                Traceback (most recent call last)

        <ipython-input-53-e894f93573ea> in <module>
    ----> 1 word[42]  # The word only has 6 characters.


        IndexError: string index out of range
```

Python strings are immutable, which means they cannot be changed. Therefore, assigning a value to an indexed position in a string results in an error:

[54]:
```
word[0] = 'J'
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-54-91a956888ca7> in <module>
    ----> 1 word[0] = 'J'


        TypeError: 'str' object does not support item assignment
```

The built-in function `len()` returns the length of a string:

[55]:
```
s = 'supercalifragilisticexpialidocious'
len(s)
```

[55]:
```
34
```

A couple of other useful things for strings are changing to upper or lower case:

[56]:
```
s = 'Python'
s.upper()
```

```
[56]: 'PYTHON'
```

```
[57]: s.lower()
```

```
[57]: 'python'
```

You can also test if a part of a string is inside a larger string (more on conditionals later):

```
[58]: 'thon' in 'Python'
```

```
[58]: True
```

```
[59]: 'blah' in 'Python'
```

```
[59]: False
```

We can use slicing to reverse a string:

```
[60]: s[::-1]
```

```
[60]: 'nohtyP'
```

We can split a string into parts based on a particular character:

```
[61]: s = 'milk, eggs, chocolate, ice cream, bananas, cereal, coffee'
      s.split(',')
```

```
[61]: ['milk', ' eggs', ' chocolate', ' ice cream', ' bananas', ' cereal', ' coffee']
```

The above cut the string up into a smaller string each time it encountered a comma. The results is a list… speaking of!

1.8 Collections

Everything we have considered so far is mostly single elements (numbers, strings). However, we often also need to deal with collections of numbers and things (actually a, string can be thought of as a collection of individual characters).

There are three built-in types of collections that are useful to know about for this class: - lists - dictionaries - sets

1.8.1 Lists

Python knows a number of *compound* data types, which are used to group together other values. The most versatile is the *list*, which can be written as a sequence of comma-separated values (items) between square brackets.

Lists might contain items of different types, but usually the items all have the same type.

Here is an empty list that contains nothing:

```
[62]: squares = []
      squares
```

```
[62]: []
```

Here is a list of numbers

```
[63]: squares = [1, 4, 9, 16, 25]
      squares
```

```
[63]: [1, 4, 9, 16, 25]
```

and here are two lists of either all strings or a mixture of numbers and strings:

```
[64]: squares_string = ["one", "four", "nine", "sixteen", "twentyfive"]
      squares_string
```

```
[64]: ['one', 'four', 'nine', 'sixteen', 'twentyfive']
```

```
[65]: squares_mixed = ["one", 4, 9, "sixteen", 25]
      squares_mixed
```

```
[65]: ['one', 4, 9, 'sixteen', 25]
```

Like strings (and all other built-in sequence types), lists can be indexed and sliced:

```
[66]: squares[0]  # Indexing returns the item.
```

```
[66]: 1
```

```
[67]: squares[-1]
```

```
[67]: 25
```

```
[68]: squares[-3:]  # Slicing returns a new list.
```

```
[68]: [9, 16, 25]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a new (shallow) copy of the list:

```
[69]: squares[:]
```

```
[69]: [1, 4, 9, 16, 25]
```

Lists also support concatenation with the + operator:

```
[70]: squares + [36, 49, 64, 81, 100]
```

```
[70]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, which means you can change any value in the list:

```
[71]: cubes = [1, 8, 27, 65, 125]  # Something's wrong here ...
      4 ** 3  # the cube of 4 is 64, not 65!
```

```
[71]: 64
```

```
[72]: cubes[3] = 64  # Replace the wrong value.
      cubes
```

```
[72]: [1, 8, 27, 64, 125]
```

Use the list's `append()` method to add new items to the end of the list:

```
[73]: cubes.append(216)  # Add the cube of 6 ...
      cubes.append(7 ** 3)  # and the cube of 7.
      cubes
```

```
[73]: [1, 8, 27, 64, 125, 216, 343]
```

You can even assign to slices, which can change the size of the list or clear it entirely:

```
[74]: letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
      letters
```

```
[74]: ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

```
[75]: # Replace some values.
      letters[2:5] = ['C', 'D', 'E']
      letters
```

```
[75]: ['a', 'b', 'C', 'D', 'E', 'f', 'g']
```

```
[76]: # Now remove them.
      letters[2:5] = []
      letters
```

```
[76]: ['a', 'b', 'f', 'g']
```

```
[77]: # Clear the list by replacing all the elements with an empty list.
      letters[:] = []
      letters
```

```
[77]: []
```

The built-in `len()` function also applies to lists:

```
[78]: letters = ['a', 'b', 'c', 'd']
      len(letters)
```

```
[78]: 4
```

You can nest lists, which means to create lists that contain other lists. For example:

```
[79]: a = ['a', 'b', 'c']
      n = [1, 2, 3]
      x = [a, n]
      x
```

```
[79]: [['a', 'b', 'c'], [1, 2, 3]]
```

```
[80]: x[0]
```

```
[80]: ['a', 'b', 'c']
```

```
[81]: x[0][1]
```

```
[81]: 'b'
```

You can create lists from scratch using a number of methods. For example, to create a list containing all the numbers from 0 to 10, you can use the `range()` function which automatically generates an iterator that steps through a set of values:

```
[82]: range(10)
```

```
[82]: range(0, 10)
```

```
[83]: type(range(10))
```

```
[83]: range
```

Let's convert the iterator to list to see the values:

```
[84]: list(range(10))
```

```
[84]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note that we do not need to convert an iterator to list if we just want to work with it, e.g., loop over the values and print them:

```
[85]: for integer in range(10):
          print(integer)
```

```
0
1
2
```

```
3
4
5
6
7
8
9
```

You can also create lists by repeating a list many times:

```
[86]: [1] * 10
```

```
[86]: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
[87]: [0] * 10
```

```
[87]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
[88]: [1,2] * 10
```

```
[88]: [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

Finally, you can create a list by means of list comprehension (a form of lambda / set abstraction):

```
[89]: # cubes from 1 to 10
      [n**3 for n in range(1, 11)]
```

```
[89]: [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

1.8.2 Dictionaries

Dictionaries are collections of `key-value` pairs.

One easy way to understand the difference between lists and dictionaries is that:

- in a list, you can "lookup" an entry using a index value (e.g., `squares[1]`)
- in a dictionary, you can look up values using anything as the index value, including string, numbers, or other Python elements (technically any object which is hashable).

First, we can create an empty dictionary that contains nothing:

```
[90]: person = {}
```

Next, we might want to try initializing with some key value pairs.

- key-value pairs are separated by commas (similar to a list collection)
- but the key and value are separated with `:`

```
[91]: person = {'firstname': 'Adrian', 'lastname': 'Brasoveanu', 'office': 259,␣
       ↪'building': 'Stevenson'}
```

```
person
```

```
[91]: {'firstname': 'Adrian',
       'lastname': 'Brasoveanu',
       'office': 259,
       'building': 'Stevenson'}
```

The main think about dictionaries is that you can "lookup" any value you want by the "key":

```
[92]: person['firstname']
```

```
[92]: 'Adrian'
```

```
[93]: person['office']
```

```
[93]: 259
```

You can also create a dictionary using the `dict()` function (just like you can create a list using the `list()` function):

```
[94]: person = dict('firstname': 'Adrian', 'lastname': 'Brasoveanu', 'office': 259,␣
       ↪'building': 'Stevenson'}
      person
```

```
          File "<ipython-input-94-a17be96d2db5>", line 1
        person = dict('firstname': 'Adrian', 'lastname': 'Brasoveanu', 'office':␣
  ↪259, 'building': 'Stevenson'}
                                 ^
    SyntaxError: invalid syntax
```

Sometimes you want to look inside a dictionary to see all the elements.

This is all the keys:

```
[95]: person.keys()
```

```
[95]: dict_keys(['firstname', 'lastname', 'office', 'building'])
```

Values:

```
[96]: person.values()
```

```
[96]: dict_values(['Adrian', 'Brasoveanu', 259, 'Stevenson'])
```

Items, i.e., the actual key-value pairs:

```
[97]: person.items()
```

```
[97]: dict_items([('firstname', 'Adrian'), ('lastname', 'Brasoveanu'), ('office',
      259), ('building', 'Stevenson')])
```

Looking up a key that doesn't exists in the dictionary results in a `KeyError` error:

```
[98]: print(person['address'])
```

```
        ---------------------------------------------------------------------------

        KeyError                                  Traceback (most recent call last)

        <ipython-input-98-b106afb068c0> in <module>
    ----> 1 print(person['address'])


        KeyError: 'address'
```

In addition to initializing a dictionary you can add to it later:

```
[99]: person = {'firstname': 'Adrian', 'lastname': 'Brasoveanu', 'office': 259,␣
       ↪'building': 'Stevenson'}
      person['age']='>30'
      person['position']='professor'
      person['email']='abrsvn@ucsc.edu'
      person
```

```
[99]: {'firstname': 'Adrian',
       'lastname': 'Brasoveanu',
       'office': 259,
       'building': 'Stevenson',
       'age': '>30',
       'position': 'professor',
       'email': 'abrsvn@ucsc.edu'}
```

You can also overwrite an existing value:

```
[100]: person['firstname']='UNKNOWN'
       person
```

```
[100]: {'firstname': 'UNKNOWN',
        'lastname': 'Brasoveanu',
        'office': 259,
        'building': 'Stevenson',
        'age': '>30',
        'position': 'professor',
        'email': 'abrsvn@ucsc.edu'}
```

In addition to indexing by key using the `[]`, you can use the `.get()` function to lookup by a key. This is useful because you can provide an optional value in case the lookup fails:

[101]:
```
# this works
person.get('office')
```

[101]:
```
259
```

[102]:
```
# this fails but you get to return 'oops' in that case instead of an error
person.get('phonenumber', 'UNKNOWN')
```

[102]:
```
'UNKNOWN'
```

You can merge two dictionary together:

[103]:
```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3}
dict1.update(dict2)
print(dict1)
```

```
{'a': 1, 'b': 2, 'c': 3}
```

If they have the same keys, the second one will overwrite the first.

[104]:
```
# If they have same keys:
dict1.update({'c': 4})
print(dict1)
```

```
{'a': 1, 'b': 2, 'c': 4}
```

Dictionaries are a very useful and fast way of organizing data, and all modern computer languages have a hash table data structure like this.

For example, one might naturally think of the columns of a excel spreadsheet or data file as being labeled with 'keys' that have a list of values underneath them.

This is exactly a data format that `pandas` (a library that we will use in this class; more on this and other libraries later) likes:

[105]:
```
import pandas as pd

df = pd.DataFrame({'student': [1,2,3,4], 'grades': [0.95, 0.27, 0.45, 0.8]})
df
```

[105]:
```
   student  grades
0        1    0.95
1        2    0.27
2        3    0.45
3        4    0.80
```

Another reason is that a very common data file format online is JSON, which is a data file format composed of key-value pairs. (as a matter of fact, this notebook is underlying a JSON file)

Here is an example of a string that contains JSON:

```
[106]: jsonstring='''
{
     "firstName": "John",
     "lastName": "Smith",
     "address": {
         "streetAddress": "21 2nd Street",
         "city": "New York",
         "state": "NY",
         "postalCode": 10021
     },
     "phoneNumbers": [
         "212 555-1234",
         "646 555-4567"
     ]
 }
'''
```

Which can be quickly loaded into a dictionary using the `json` library:

```
[107]: import json

json_dictionary = json.loads(jsonstring)
json_dictionary
```

```
[107]: {'firstName': 'John',
 'lastName': 'Smith',
 'address': {'streetAddress': '21 2nd Street',
  'city': 'New York',
  'state': 'NY',
  'postalCode': 10021},
 'phoneNumbers': ['212 555-1234', '646 555-4567']}
```

```
[108]: json_dictionary['firstName']
```

```
[108]: 'John'
```

```
[109]: json_dictionary['phoneNumbers']
```

```
[109]: ['212 555-1234', '646 555-4567']
```

```
[110]: json_dictionary['address']
```

[110]:
```
{'streetAddress': '21 2nd Street',
 'city': 'New York',
 'state': 'NY',
 'postalCode': 10021}
```

This last example shows how a dictionary can be a value inside a dictionary.

1.8.3 Sets

The final collection we will discuss is a set.

You might have learned about sets and set theory in high school. A set is an unordered collection of objects so, unlike a list, there is not a "first element" or a "second element" in a set. Instead, a set just contains objects and allows you to do various types of set operations, such as testing if an element is within a set, testing if a set is a subset of another set, etc.

You can create a set like this:

[111]:
```
animal_set = set(['Coyote', 'Dog', 'Bear', 'Cat', 'Elephant'])
animal_set
```

[111]:
```
{'Bear', 'Cat', 'Coyote', 'Dog', 'Elephant'}
```

Notice how even though 'Coyote' was the first element in the list that we used to initialize the set, it is no longer the first item in the output. This is because sets are **unordered**, and Python lists set elements in an arbitrary (in this case, alphabetic) order.

Sets are closely related to dictionaries. Just like the elements of a dictionary, sets are unordered (and set elements have to be hashable, just like dict keys). But dicts are indexed by keys, whereas a set is **unindexed**.

Sets differ from lists in two ways, as lists are both ordered and indexed. But just like lists, sets are mutable.

Fyi: tuples are a 4th built-in collection/container that are ordered and indexed like a list, but are immutable. - For example, when the items of a dict are listed, each item is a tuple

[112]:
```
list(person.items())[0]
```

[112]:
```
('firstname', 'UNKNOWN')
```

[113]:
```
type(list(person.items())[0])
```

[113]:
```
tuple
```

Because of their close connection to dicts, we can create sets using `{}`:

[114]:
```
set_1 = {2, 4, 6, 8, 10}
set_2 = {42, 'foo', (1, 2, 3), 3.14159}
type(set_1), type(set_2)
```

[114]:
```
(set, set)
```

The size of a set can be found with the `len()` operator we saw before:

[115]:
```
len(animal_set)
```

[115]:
```
5
```

You can test if an element is contained within a set:

[116]:
```
'owl' in animal_set
```

[116]:
```
False
```

[117]:
```
'Coyote' in animal_set
```

[117]:
```
True
```

You can take the union of two sets, which will find all the elements unique and in common to both set. Repeats are removed this way:

[118]:
```
x1 = {'foo', 'bar', 'baz'}
x2 = {'baz', 'qux', 'quux'}
```

[119]:
```
x1 | x2
```

[119]:
```
{'bar', 'baz', 'foo', 'quux', 'qux'}
```

[120]:
```
# or
x1.union(x2)
```

[120]:
```
{'bar', 'baz', 'foo', 'quux', 'qux'}
```

You can also find the set intersection

[121]:
```
x1 = {'foo', 'bar', 'baz'}
x2 = {'baz', 'qux', 'quux'}
```

[122]:
```
x1.intersection(x2)
```

[122]:
```
{'baz'}
```

[123]:
```
# or
x1 & x2
```

[123]:
```
{'baz'}
```

The set-difference operator finds the things in one set that are not in the other:

```
[124]: x1 = {'foo', 'bar', 'baz'}
       x2 = {'baz', 'qux', 'quux'}

       x1.difference(x2)
```

```
[124]: {'bar', 'foo'}
```

```
[125]: # or
       x1 - x2
```

```
[125]: {'bar', 'foo'}
```

The `symmetric_difference` operator finds the things in either set A or set B but not both:

```
[126]: x1 = {'foo', 'bar', 'baz'}
       x2 = {'baz', 'qux', 'quux'}

       x1.symmetric_difference(x2)
```

```
[126]: {'bar', 'foo', 'quux', 'qux'}
```

```
[127]: # or
       x1 ^ x2
```

```
[127]: {'bar', 'foo', 'quux', 'qux'}
```

we can check if two sets have any elements in common using `isdisjoint()`.

```
[128]: x1 = {1, 3, 5}
       x2 = {2, 4, 6}

       x1.isdisjoint(x2)
```

```
[128]: True
```

```
[129]: x1 = {1, 3, 5, 7} # adding one element in commont
       x2 = {2, 4, 6, 7}

       x1.isdisjoint(x2)
```

```
[129]: False
```

We can check if one set is a subset of another one:

```
[130]: x1 = {1, 3, 5, 7}
       x2 = {2, 4, 6, 7}

       x1.issubset(x2)
```

```
[130]: False
```

```
[131]: x1 = {2,4}
       x2 = {2, 4, 6, 7}

       x1.issubset(x2)
```

```
[131]: True
```

To add or remove a new element to the set, use the `add()` or `remove()` method:

```
[132]: x1.add('foo')
       x1
```

```
[132]: {2, 4, 'foo'}
```

```
[133]: x1.remove('foo')
       x1
```

```
[133]: {2, 4}
```

Sets can be used to get all the unique elements of a list.

For example, if you have a list of ages of participants in an experiment, a set could be a nice way to find the different values it takes:

```
[134]: ages = [14, 15, 35, 15, 24, 14, 17, 18, 22, 22, 24]
       len(ages)
```

```
[134]: 11
```

```
[135]: set(ages)
```

```
[135]: {14, 15, 17, 18, 22, 24, 35}
```

```
[136]: len(set(ages))
```

```
[136]: 7
```

Thus, there are 7 unique ages in the data set.

1.9 Flow Control

Up until now, we have always executed lines of code in sequence, one after the other. But we often need to exert control over which bits of code should run depending on other variables or settings.

To do this, we use control flow expressions. These include: - conditionals like `if`, `else`, and `elif` - `for` or `while` loops (we won't talk about `while` loops, although they can be really useful)

1.9.1 Testing if things are true

The first thing we need is testing if a condition is satisfied. One of the most common ways to do this is with the double equals sign ==.

Note that the double equals sign is different from the single equals sign: - the latter **assigns** a value to a variable - the former **tests** if two things are equal or not

```
[137]: 1 == 2
```

```
[137]: False
```

```
[138]: 1 == 1
```

```
[138]: True
```

```
[139]: 'hello' == 'hello'
```

```
[139]: True
```

```
[140]: 'hello' == 'HELLO'
```

```
[140]: False
```

A common use case is testing if a variable is equal to a particular value:

```
[141]: myvar = 10

       myvar == 10
```

```
[141]: True
```

```
[142]: myvar == 11
```

```
[142]: False
```

There are related comparisons. For example, we might want to test if a variable is greater than a particular value

```
[143]: myvar > 10 # is myvar greater than 10
```

```
[143]: False
```

```
[144]: myvar >=5  # is myvar greater than or equal to 5
```

```
[144]: True
```

```
[145]: myvar <= 15 # is myvar less than or equal to 15
```

[145]: True

We can also test if the value of variable is **not** equal to a value:

```
[146]: myvar != 5
```

```
[146]: True
```

```
[147]: myvar != 10
```

```
[147]: False
```

There is also a slightly more English-like version of this test:

We can also test if an item is a part of a collection (e.g., a list or set):

```
[148]: mylist = ['one', 'two', 'three', 'four']

       'one' in mylist # is 'one' in mylist?
```

```
[148]: True
```

```
[149]: 'six' not in mylist # is 'six' not in mylist?
```

```
[149]: True
```

```
[150]: mystring = 'lkjasldfkj'

       'jas' in mystring # tests if 'jas' is a substring of mystring
```

```
[150]: True
```

```
[151]: 'jas' not in mystring # tests if 'jas' is not a substring of mystring
```

```
[151]: False
```

1.9.2 Conditionals (if-then-else)

Now that we have seen a few ways to test if things are true/false or meet some particular conditiiton, the next step is to execute different code depending on what the test gives us.

The simplest kind of expression that does this has the general form:

```
if <expression>:
    <code block>
```

where `expression` is true/false (Boolean; or can be covertly coerced into a Boolean), and `<code block>` is some collection of lines that will be executed only if the expression is `True`.

[152]:
```python
weather = 'sunny'

if weather == 'sunny':
    print('Walk the dog')
    print('Mow the lawn')
    print('Weed the flower bed')
```

```
Walk the dog
Mow the lawn
Weed the flower bed
```

[153]:
```python
weather = 'rainy'

if weather == 'sunny':
    print('Walk the dog')
    print('Mow the lawn')
    print('Weed the flower bed')
```

The important thing about this code is that you could run it even if you weren't sure what the value of `weather` was because it was set earlier in the program or by some other piece of complex code. Thus, it lets you run a special bit of code depending on the value of a variable.

Here are a couple of **very important** but sometimes **subtle or confusing** things to note about this.

First, you'll notice that the `print()` lines are not aligned with the rest of the code in that cell.

This is because the first character of that line is the tab character, or more precisely, 4 white spaces. In Python, spacing is **very important** as it it determines scope. Any line that is tabbed over from the line above it is known as a "code block":

[154]:
```python
myvar = 10
myvar2 = 20

if myvar == myvar2:
    # this is inside the code block
    print("this is")
    print("a code block")

# this is not in the code block because it is not tabbed over
print("this is not")
```

```
this is not
```

All programming languages have some type of code block / scope marking syntax, but in Python, you just use white space to do this (this is part of why Python code is so readable).

But this simplicity and readability can sometimes be confusing to new users because you really have to keep track of the level of indentation of your code.

ALWAYS indent your code by 4 white spaces; not 1, not 3, not 5. The magic number that will ensure you won't lose any points is 4.

[155]:
```
myvar = 10
myvar2 = 10

if myvar == myvar2:
    print("this is")
    print("a code block")

print("this is not")
```

```
this is
a code block
this is not
```

Consider the two contrasting examples above (the one just now and the one before it).

- in the earlier one, `myvar` and `myvar2` have a different value, so the == test fails (return `False`) and then the indented code block is skipped
- in the second example, the value of the test is `True`, so the code block is run

In both cases, the final print (which is *not* indented) runs no matter what, so it is printed out in both examples).

The general structure is this:

In the `if` statement we just considered you take an optional path through the code and then continue.

However, othertimes you want to take one path if something is `True` and another path if it is `False`. For example:

```
if raining:
   - take umbrella
otherwise:
   - take sunglasses
- take wallet
```

This is not valid Python but it makes intuitive sense… sometimes if the conditional is false we want to do something else. In Python, this is accomplished with the `else` command:

[156]:
```
raining = True

if raining:
    print("Take umbrella")
else:
    print("Take sunglasses")

print("Take wallet")
```

```
Take umbrella
Take wallet
```

[157]:
```
raining = False

if raining:
    print("Take umbrella")
else:
    print("Take sunglasses")

print("Take wallet")
```

```
Take sunglasses
Take wallet
```

If you compare the two code cells above, you can see that depending on the value of raining (either `True` or `False`... these are special words in Python), we take a different path through the code.

But what if there many conditions? In that case, we can use the `elif` (which is a combination of the `else` and `if`):

[158]:
```
weather = 'sunny'

if weather == 'raining':
    print("Take umbrella")
elif weather == 'sunny':
    print("Take sunglasses")
elif weather == 'cloudy':
    print("Take sweater")

print("Take wallet")
```

```
Take sunglasses
Take wallet
```

[159]:
```
weather = 'raining'

if weather == 'raining':
    print("Take umbrella")
elif weather == 'sunny':
    print("Take sunglasses")
elif weather == 'cloudy':
    print("Take sweater")

print("Take wallet")
```

```
Take umbrella
Take wallet
```

[160]:
```
weather = 'cloudy'

if weather == 'raining':
    print("Take umbrella")
elif weather == 'sunny':
    print("Take sunglasses")
elif weather == 'cloudy':
    print("Take sweater")

print("Take wallet")
```

```
Take sweater
Take wallet
```

If needed, you can always end a `if/elif` sequence with an final `else`:

[161]:
```
weather = 'heat wave'

if weather == 'raining':
    print("Take umbrella")
elif weather == 'sunny':
    print("Take sunglasses")
elif weather == 'cloudy':
    print("Take sweater")
else:
    print("I don't know what to do!")

print("Take wallet")
```

```
I don't know what to do!
Take wallet
```

1.9.3 For Loops

Using loops allows us to automate and repeat similar tasks multiple times.

A `for` loop implements the repeated execution of code based on a loop counter or loop variable.

This means that `for` loops are used most often when the number of repetitions is known before entering the loop, unlike **while loops**, which can run until some condition is met.

In Python, `for` loops are constructed like so:

```
for [iterating variable] in [iterable, e.g., sequence]:
    [do something]
```

The something that is being done (known as a code block) will be executed until the sequence is over. The code block itself can consist of any number of lines of code, as long as they are indented properly.

Let's look at a `for` loop that iterates through a range of values:

[162]:
```
for i in range(0,5):
   print(i)
```

```
0
1
2
3
4
```

- this `for` loop sets up `i` as its iterating variable, and the iterable is the range 0 to 5
- within the loop, we print out one integer per loop iteration

For Loops using `range()` One of Python's built-in immutable sequence types is `range()`. In loops, `range()` is used to control how many times the loop will be repeated.

When working with `range()`, you can pass between 1 and 3 integer arguments to it:

- `start` states the integer value at which the sequence begins, if this is not included then start begins at 0
- `stop` is always required and is the integer that is counted up to but not included
- `step` sets how much to increase (or decrease in the case of negative numbers) the next iteration, if this is omitted then step defaults to 1

We'll look at some examples of passing different arguments to `range()`.

First, let's only pass the `stop` argument, so that our sequence set up is `range(stop)`:

```
[163]: for i in range(6):
          print(i)
```

```
0
1
2
3
4
5
```

In the program above, the stop argument is 6, so the code will iterate from 0-6 (exclusive of 6).

Next, we'll look at `range(start, stop)`, with arguments specifying when the iteration should start and when it should stop. Below, the range goes from 20 (inclusive) to 25 (exclusive):

```
[164]: for i in range(20,25):
           print(i)
```

```
20
21
22
23
24
```

The step argument of `range()` can be used to skip values within the sequence.

With all three arguments, `step` comes in the final position: `range(start, stop, step)`. First, let's use a `step` with a positive value. In this case, the `for` loop is set up so that the numbers from 0 to

15 print out, but at a step of 3, so that only every third number is printed, like so:

```
[165]: for i in range(0,15,3):
           print(i)
```

```
0
3
6
9
12
```

We can also use a negative value for our `step` argument to iterate backwards, but we'll have to adjust our start and stop arguments accordingly. Below, 100 is the `start` value, 0 is the `stop` value, and -10 is the range, so the loop begins at 100 and ends at 0, decreasing by 10 with each iteration:

```
[166]: for i in range(100,0,-10):
           print(i)
```

```
100
90
80
70
60
50
40
30
20
10
```

For Loops using Sequential Data Types Lists and other data types can be used as iterables. Rather than iterating through a `range()`, you can define a list and iterate through that list.

We'll assign a list to a variable, and then iterate through the list:

```
[167]: sharks = ['hammerhead', 'great white', 'dogfish', 'frilled', 'bullhead',␣
        ↪'requiem']

       for shark in sharks:
           print(shark)
```

```
hammerhead
great white
dogfish
frilled
bullhead
requiem
```

The output above shows that the `for` loop iterated through the list, and printed each item from the list per line.

We can iterate over a set too (or a tuple):

[168]:
```
sharks = {'hammerhead', 'great white', 'dogfish', 'frilled', 'bullhead', 
 ↪'requiem'}

for shark in sharks:
   print(shark)
```

```
requiem
dogfish
bullhead
hammerhead
frilled
great white
```

Lists and other sequence-based data types like strings and tuples are common to use with loops because they are iterable.

You can combine these data types with range() to add items to a list, for example:

[169]:
```
sharks = ['hammerhead', 'great white', 'dogfish', 'frilled', 'bullhead', 
 ↪'requiem']

for item in range(len(sharks)):
   sharks.append('shark')

print(sharks)
```

```
['hammerhead', 'great white', 'dogfish', 'frilled', 'bullhead', 'requiem',
'shark', 'shark', 'shark', 'shark', 'shark', 'shark']
```

You can also use a `for` loop to construct a list from scratch. In this example, the list `integers` is initialized as an empty list, but the for loop populates the list like so:

[170]:
```
squares = []

for i in range(10):
   squares.append(i**2)

print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In some cases – like the one above – list comprehensions are more readable (and faster) than for loops:

[171]:
```
squares = [i**2 for i in range(10)]

print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Similarly, we can iterate through strings:

[172]:
```python
sammy = 'Sammy'

for letter in sammy:
   print(letter)
```

```
S
a
m
m
y
```

When iterating through a dictionary, it's important to keep the `key:value` structure in mind to ensure that you are calling the correct element of the dictionary. Here is an example that calls both the key and the value:

[173]:
```python
sammy_shark = {'name': 'Sammy', 'animal': 'shark', 'color': 'blue', 'location':␣
 ↪'ocean'}

for key in sammy_shark:
   print(key + ': ' + sammy_shark[key])
```

```
name: Sammy
animal: shark
color: blue
location: ocean
```

When using dictionaries with `for` loops, the iterating variable corresponds to the keys of the dictionary, and `dictionary_variable[iterating_variable]` corresponds to the values.

In the case above, the iterating variable key was used to stand for `key`, and `sammy_shark[key]` was used to stand for the values.

Loops are often used to iterate over and manipulate sequential data types.

Nested For Loops Loops can be nested in Python, as they can with other programming languages.

A nested loop is a loop that occurs within another loop, structurally similar to nested if statements. These are constructed like so:

```
for [first iterating variable] in [outer loop]: # Outer loop
    [do something]  # Optional
    for [second iterating variable] in [nested loop]:   # Nested loop
        [do something]
```

- The program first encounters the outer loop, executing its first iteration.
- This first iteration triggers the inner, nested loop, which then runs to completion.

- Then the program returns back to the top of the outer loop, completing the second iteration and again triggering the nested loop.
- Again, the nested loop runs to completion, and the program returns back to the top of the outer loop until the sequence is complete or a break or other statement disrupts the process.

Let's implement a nested `for` loop so we can take a closer look. In this example, the outer loop will iterate through a list of integers called `num_list`, and the inner loop will iterate through a list of strings called `alpha_list`.

[174]:
```
num_list = [1, 2, 3]
alpha_list = ['a', 'b', 'c']

for number in num_list:
    print(number)
    for letter in alpha_list:
        print(letter)
    print('--')
```

```
1
a
b
c
--
2
a
b
c
--
3
a
b
c
--
```

The output illustrates that the program completes the first iteration of the outer loop by printing `1`, which then triggers completion of the inner loop, printing `a`,`b`, `c` consecutively.

Once the inner loop has completed, the program prints a separator `--` and returns to the top of the outer loop, prints `2`, then again prints the inner loop in its entirety (`a`, `b`, `c`), etc.

Nested `for` loops can be useful for iterating through items within lists composed of lists. In a list composed of lists, if we employ just one for loop, the program will output each internal list as an item:

[175]:
```
list_of_lists = [['hammerhead', 'great white', 'dogfish'],
                 [0, 1, 2],
                 [9.9, 8.8, 7.7]]

for list in list_of_lists:
    print(list)
```

```
['hammerhead', 'great white', 'dogfish']
[0, 1, 2]
[9.9, 8.8, 7.7]
```

In order to access each individual item of the internal lists, we'll use a nested `for` loop:

[176]:
```
list_of_lists = [['hammerhead', 'great white', 'dogfish'],[0, 1, 2],[9.9, 8.8, 7.
 ↪7]]

for list in list_of_lists:
    for item in list:
        print(item)
    print('------')
```

```
hammerhead
great white
dogfish
------
0
1
2
------
9.9
8.8
7.7
------
```

The `while`-loop is another useful looping structure. Part of your homework for next class is to watch this great tutorial on RealPython.

1.10 Writing New Functions

In the previous sections, we learned how we can organize code into **code blocks** by indenting it and wrapping it in control structures, which then execute the code a certain number of times or depending on some particular condition.

As you begin to write longer and longer programs though, it sometimes helps to break up the functionality of your programs into reuseable chunks called **functions**. Functions (and classes) make your code more readable and more modular.

The general format for a function is composed of a few elements.

- First there is a line the defines the **name** of the function and the **parameters** is can take (more on that later).
- Next, there is a sequence of instructions that are indented; this is the body of the function

```
def my_function(parameter1, parameter2):
    <code line 1>
    <code line 2>
    <code line 3>
```

Here's an example:

[177]:
```
def my_function():
    print("Hello from my function")
```

Notice two things about this function:

- First is that is has no parameters (which is fine… we'll talk about how to add them later).
- Second, when you run this code cell, nothing happens; this is because we simply **defined** the function, but we did not run it

To run this function, we write:

[178]:
```
my_function()
```

```
Hello from my function
```

Here we "executed" (i.e., ran) the function by calling its name **with the parentheses**.

If the parentheses are missing, the function is not called: we simply ask Python what object is assigned to the variable. The object is a function:

[179]:
```
my_function
```

[179]:
```
<function __main__.my_function()>
```

Functions can be combined with other elements of Python to create more complex program flows. For example, we can combine a custom function with a for loop:

[180]:
```
def my_ramp():
    print('*')
    print('***')
    print('*****')

for i in range(4):
    my_ramp()
    print('------')
```

```
*
***
*****
------
*
***
*****
------
*
***
*****
------
*
```

```
***
*****
------
```

The above code, which prints out a ramping set of stars four times, is much more efficient than if you copy-pasted the print statements 4 times.

Parameters Functions can take different types of "parameters", which are essentially variables that are created at the start of the execution of the function. This is helpful for making more abstract functions that perform some computation with their inputs. For example:

[181]:
```python
def add(x, y):
    print("The sum is:", x + y)
```

The `add` function above takes two parameters, or arguments, `x` and `y`.

- These parameters are then used as the operands to an addition operation.

As a result, we can run this function many times with different inputs and get different results:

[182]:
```python
add(1, 2)
```

```
The sum is: 3
```

[183]:
```python
add(34, 28)
```

```
The sum is: 62
```

Return values The functions we have considered so far simply print something out and then finish. However, sometimes you might like to have your function give back one or more result for further processing by other parts of your program. For instance, instead of printing out the sum of x and y, we can redefine the `add()` function to return the sum using a special keyword `return`:

[184]:
```python
def add(x,y):
    return x + y
```

Now when we run this function, instead of printing out a message, the value of the sum is calculated and passed back.

[185]:
```python
add(4, 5)
```

[185]:
```
9
```

This allows you to continue to do additional processing. For example, we can add two numbers and then compute the square root of them:

[186]:
```python
x = add(4, 5)
x**2
```

[186]:
```
81
```

You can return multiple values from a function if you want. For example, we could create a function call `arithmetic` that does a number of operations to x and y and returns them all:

```
[187]: def arithmetic(x, y):
           _sum = x + y
           _diff = x - y
           _prod = x * y
           _div = x / y
           return _sum, _diff, _prod, _div
```

This new function gives back a `tuple` that contains all four of the results in one step.

```
[188]: arithmetic(7, 3)
```

```
[188]: (10, 4, 21, 2.3333333333333335)
```

1.11 Importing additional functionality

One of the best features of Python is the large number of add-on packages.

For example, there are packages for basic data analysis (`pandas`), plotting (`matplotlib` or `seaborn`), we'll use a special package / library called `pyactr` in this class, etc.

These packages are not all loaded automatically. Instead at the start of your notebook or program, you often need to **import** this functionality.

There are a couple of ways to import packages.

Basic importing is accomplished with the `import` command. For example, to import the `math` module in Python, we just type:

```
[189]: import math
```

Now we can access the methods provided by the math function using the `.` (dot) operator. Any function that the `math` library provides is accessible to us now using `math.<function>`. For example, to compute the cosine of a number:

```
[190]: math.cos(1)
```

```
[190]: 0.5403023058681398
```

Sometimes we want to import a library but rename it so that it is easier to type. Many popular packages are imported using the `import <package> as <shortname>` syntax. This imports the library but immediately changes its name, usually to something simpler and shorter for easy typing.

For example it is traditional to import pandas and rename it `pd`:

```
[191]: import pandas as pd
```

Now we have access to the pandas function using `pd.<something>`. For example, we can create a new pandas *dataframe* like this:

```
[192]: df = pd.DataFrame({'student': [1,2,3,4], 'grades':[0.95, 0.27, 0.45, 0.8] })
       df
```

```
[192]:    student  grades
       0        1    0.95
       1        2    0.27
       2        3    0.45
       3        4    0.80
```

If you know there is a particular function you want to grab from a library, you can import only that function. For example, we could import the `cos()` function from the `math` library like this:

```
[193]: from math import cos
```

This is generally useful if there is a really large and complex library and you only need part of it for your work.

Python libraries can really extend your computing and data options. For example, one cool library is the `wikipedia` library which provides an interface to the Wikipedia site.

```
[194]: # if on google colab, you'll have to install the library first by uncommenting␣
        ↪the line below
       # !pip3 install wikipedia
```

```
[195]: import wikipedia as wk

       wk.search("Carbon Dioxide")
```

```
[195]: ['Carbon dioxide',
        'Carbon dioxide scrubber',
        'Carbon dioxide removal',
        'Carbon capture and storage',
        'Carbon sequestration',
        'Liquid carbon dioxide',
        "Carbon dioxide in Earth's atmosphere",
        'Carbon dioxide laser',
        'Supercritical carbon dioxide',
        'Carbon sink']
```

The `search()` method will return all the wikipedia pages that match a particular search string.

1.12 Dealing with error messages

Often times you will run into error messages when using Python. This is totally fine! It is not a big deal and comes up all the time. In fact, if you don't make errors, you are probably not tinkering and learning enough.

Most errors in python generate what is known as an "exception" where the code doesn't necessarily "crash" but a warning is issued. For example, if we have too many parentheses (each parentheses you open must be closed by the same type of character) you will get a `SyntaxError`:

```
[196]: print(0/0))
```

```
      File "<ipython-input-196-ded43ae9deb7>", line 1
    print(0/0))
              ^
SyntaxError: unmatched ')'
```

There are a couple of things to notice about this error. First is that it does give you some indication of what happened (for example, it says "SyntaxError", which means the code doesn't look right to python and so it can't understand it).

It also shows you where the first error occured (here on line 1 of the cell), and it even indicates which character in that line might be the problem.

Based on this error message, you could easily fix your code by removing the extra parens at the end:

```
[197]: print(0/0)
```

```
    ---------------------------------------------------------------------------

    ZeroDivisionError                         Traceback (most recent call last)

    <ipython-input-197-1babb3b33639> in <module>
----> 1 print(0/0)


    ZeroDivisionError: division by zero
```

But this gives a different error! Now we are getting an error because we are attempting to divide by zero.

In simple cases like the one above, you can simply fix the code in the cell, try to re-run it and keep going until you get it right.

Things might not always be so easy because errors compound on one another, or arise because of the way different parts of the code interact. Reading the error messages carefully and doing some debugging might be necessary.

1.13 Further Reading and Resources

- A nice, free textbook "How to Code in Python" by Lisa Tagliaferri

- Microsoft has an Introduction to Python video series. Each video is about 10 minutes long and introduces very basic python features.
- A nice multi-part tutorial on Data Visualization with Python and Seaborn that gets into many more details about Seaborn than we have time to cover in class.
- A six hour (free) video course on basic Python programming on youtube

[ ]:

```

```