day_one

April 25, 2016

1 Day One

1.1 Table of Contents

Links to documentation will be provided at the beginning and end of each section. Look for: **DOCS**.

In today's workshop, we'll learn how to combine data types into structures and how to use them for specific purposes. We will also cover looping and interacting with operating systems. Let's get started.

1.2 Data Model

DOCS

> Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects.

Every object in Python has a **type**, a **value**, and an **identity**. We've already seen several data types, such as `int`, `float`, and `str`. An object's type determines its supported operations as well as the possible values it can take.

In some cases, an object's value can change. We call these type of objects *mutable*. Objects whose values cannot be changed are known as *immutable*. The object type determines its mutability. Numbers and strings, for example, are immutable; lists and dictionaries, which we'll cover shortly, are mutable.

To make this concrete, let's describe what an object's identity is. This can be thought of as an object's address in memory. Specifically, it's the memory address for the *value* of the object. Once an object has been created, it's identity never changes.

```
In [1]: x = 'hello'

In [2]: hex(id(x))

Out[2]: '0x1042b04c8'
```

The variable `x`'s identity or memory address is _____ (represented as a hexadecimal string). Note that the memory addresses will be different each time this code is run.

What happens if we create a new variable, `y`, and set it equal to `x`?

```
In [3]: y = x

In [4]: hex(id(y))
```

```
Out[4]: '0x1042b04c8'

In [5]: hex(id(x))

Out[5]: '0x1042b04c8'
```

The address in memory is the same because both variables *point* to (or reference) the same *value*.
Now, let's make `x` take on some other value.

```
In [6]: x = 'goodbye'

In [7]: hex(id(x))

Out[7]: '0x1042b0848'
```

Now, the address *is* different.
Let's see what happens if we set `x` to equal `hello` once more.

```
In [8]: x = 'hello'

In [9]: hex(id(x))

Out[9]: '0x1042b04c8'
```

`x` is once again pointing to the memory address associated with `hello`.

What does this have to do with mutability? It seems as though we were actually able to change `x`'s value. To answer this, we'll show an example using a mutable object—a list in this case.

```
In [10]: a = [1, 2, 3]

In [11]: hex(id(a))

Out[11]: '0x1042ce8c8'

In [12]: a.append(4)
         a

Out[12]: [1, 2, 3, 4]

In [13]: hex(id(a))

Out[13]: '0x1042ce8c8'
```

Notice what happened. We added `4` to the list, but the memory address *did not* change. This is what is means to be mutable. The value in memory address `0x107f26608` was originally `[1, 2, 3]`, but is now `[1, 2, 3, 4]`. The address in memory for this object's value will never change.

```
In [14]: a.append('#python')
         a

Out[14]: [1, 2, 3, 4, '#python']

In [15]: hex(id(a))

Out[15]: '0x1042ce8c8'
```

Now let's see what happens when we assign our list `a` to a new variable `b`.

```
In [16]: b = a
```

```
In [17]: b

Out[17]: [1, 2, 3, 4, '#python']

In [18]: hex(id(b))

Out[18]: '0x1042ce8c8'
```

That makes sense. `a` and `b` both reference the same object—`[1, 2, 3, 4, #python]`.

> Assignment statements in Python do not copy objects, they create bindings between a target and an object.

If we modify `b`, what will happen to `a`?

```
In [19]: b[-1] = 'Python'

In [20]: b

Out[20]: [1, 2, 3, 4, 'Python']

In [21]: a

Out[21]: [1, 2, 3, 4, 'Python']

In [22]: hex(id(a)) == hex(id(b))

Out[22]: True
```

The changes made to `b` have affected `a` because they both point to the same data. It's possible that this behavior is unwanted. As a solution, we can make a copy of the object so that modifying one does not affect the other. To do so, we can use the built-in `copy` module.

```
In [23]: import copy

In [24]: c = copy.copy(a)
```

This is referred to as making a *shallow* copy. While the values in `a` and `c` are the same, their respective memory addresses are different.

```
In [25]: hex(id(a)) == hex(id(c))

Out[25]: False
```

A shallow copy creates a new container (a list in this case)—which is why the addresses in memory are different—with *references* to the *contents* of the original object.

```
In [26]: hex(id(a[-1]))

Out[26]: '0x10234c1b8'

In [27]: hex(id(c[-1]))

Out[27]: '0x10234c1b8'
```

The addresses in memory for the individual elements are the same for both lists. Because we've made a copy, though, we can now modify one list without affecting the other.

```
In [28]: c[-1] = 'PYTHON'
```

```
In [29]: c

Out[29]: [1, 2, 3, 4, 'PYTHON']

In [30]: a

Out[30]: [1, 2, 3, 4, 'Python']
```

What if we were dealing with nested mutable? For this, we'll use a dictionary.

```
In [31]: d0 = {'key' : {'nested' : 'thing'}}
         d1 = copy.copy(d0)

In [32]: d1

Out[32]: {'key': {'nested': 'thing'}}

In [33]: d1['key']['nested'] = 'dict'

In [34]: d0 == d1

Out[34]: True

In [35]: d0

Out[35]: {'key': {'nested': 'dict'}}
```

Our intention was to change `d1`, but `d0` was also changed. This is because shallow copies reference contents—they don't copy them. For this, the `copy` module provides the `deepcopy()` function. Let's try that again.

```
In [36]: d0 = {'key' : {'nested' : 'thing'}}
         d1 = copy.deepcopy(d0)
         d1['key']['nested'] = 'dict'

In [37]: d0 == d1

Out[37]: False

In [38]: d0

Out[38]: {'key': {'nested': 'thing'}}

In [39]: d1

Out[39]: {'key': {'nested': 'dict'}}
```

Now that we've learned about mutability and copying objects, let's dive into data structures.
Data model **DOCS**

1.3 Data Structures

DOCS

A data structure can be thought of as a "container" for storing data that includes functions, called "methods," that are used to access and manipulate that data. Python has several built-in data structures.

1.3.1 Basics

Lists A list is a sequence of values. The values are called elements (or items) and can be of any type—integer, float, string, boolean, etc.

As a simple example, consider the following list.

```
In [40]: [1, 2, 3]

Out[40]: [1, 2, 3]
```

Notice how the list was constructed. We used square brackets around the list elements.

Let's look at a few more examples.

```
In [41]: [1.0, 8.0, 6.8]

Out[41]: [1.0, 8.0, 6.8]

In [42]: ['this', 'is', 'also', 'a', 'valid', 'list']

Out[42]: ['this', 'is', 'also', 'a', 'valid', 'list']

In [43]: [True, False, True]

Out[43]: [True, False, True]
```

It's also fine to have a list with different element types.

```
In [44]: [1, 2.0, 'three']

Out[44]: [1, 2.0, 'three']
```

Lists can even be nested—which means you can have lists within lists.

```
In [45]: [350, 'barrows', 'hall', ['berkeley', 'CA']]

Out[45]: [350, 'barrows', 'hall', ['berkeley', 'CA']]
```

This nesting can be arbitrarily deep, but it's not usually a good idea as it can get confusing. For example, it may be difficult to access specific items for an object like:

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

Speaking of accessing elements, let's describe how to do that. We'll first create a new list and assign it to a variable called `first_list`.

```
In [46]: first_list = [9, 8, 7.0, 6, 5.4]
```

To access list elements, we use the square bracket notation. For example, if we're interested in the middle element—the "two-eth" element—we use the following.

```
In [47]: first_list[2]

Out[47]: 7.0
```

This is called indexing and the value inside of the brackets must be an integer. (Recall that indices in Python start at `0`.) A list can be thought of mapping (or correspondence) between indices and elements.

Let's say you're interested in the *last* element of this list. How could you do that? If you know the length of the list, you could access it using something like:

```
first_list[len(first_list) - 1]
```

Why is the `-1` needed?

There is an easier way. Python provides negative indices that let you access elements from "back-to-front."

```
In [48]: first_list[-1]
```

```
Out[48]: 5.4
```

With this notation, the last element is accessed with `-1` (because `-0 == 0`). Use `-2` to access the second-to-last item, `-3` to access the third-to-last element, and so on.

We can also use the slice operator on lists to access multiple elements. The operator takes the following form: `[n:m]`. The first value before the colon (`:`) specifies the start position and the second value specifies the end position. The former is inclusive and the latter is exclusive. Let's take a look at what we mean.

To motivate this, let's label the indices of our list.

```
list:  [9, 8, 7.0, 6, 5.4]
index: [0, 1,   2, 3,   4]
```

The code we'll submit is: `first_list[0:2]`. This tells Python to include values associated with position 0, position 1, but **not** for position 2.

```
In [49]: first_list[0:2]
```

```
Out[49]: [9, 8]
```

This is how Python has decided to make this operator work. This isn't intuitive, but thinking about it in the following way might help. If we consider the indices to be to the *left* of each item, we can think of the slice operator as accessing elements *between* those indices.

If you try to access an item at an index that doesn't exist, Python will throw an `IndexError`:

```
In [50]: first_list[10]
```

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

IndexError                                Traceback (most recent call last)

<ipython-input-50-9b47483cd835> in <module>()
----> 1 first_list[10]


IndexError: list index out of range
```

from Raymond Hettinger

If, however, I try to access the same item with a slicing operation, e.g. `first_list[10:11]`, there is no error. Why?

```
In [51]: first_list[10:11]
```

```
Out[51]: []
```

With lists, because they are mutable, we can modify elements.

```
In [52]: first_list[-1] = 5.43
```

```
In [53]: first_list
```

```
Out[53]: [9, 8, 7.0, 6, 5.43]
```

Dictionaries A dictionary is a mapping from *keys* to *values*, where the keys, which must be unique, can be (almost) any type. A key and its associated value is referred to as a *key-value pair* or item. Dictionaries can be thought of as *unordered* key-value pairs.

There are several ways to construct a dictionary. We can use braces ({}) or the built-in `dict()` function.

```
In [54]: {}

Out[54]: {}

In [55]: dict()

Out[55]: {}
```

Of course, these are empty. Let's add comma separated key-value pairs to the first and use the assignment operator (=) for the second.

```
In [56]: {'one' : 1, 'two' : 2}

Out[56]: {'one': 1, 'two': 2}

In [57]: dict(one=1, two=2)

Out[57]: {'one': 1, 'two': 2}
```

Keys and values are themselves separated by colons.

Dictionaries are typically used for accessing values associated with keys. In the example above, we started to create a mapping between number words and their integer representations. Let's expand on this.

```
In [58]: nums = {'one' : 1, 'two' : 2, 'three' : 3, 'four' : 4, 'five' : 5, 'six' : 6}

In [59]: nums

Out[59]: {'five': 5, 'four': 4, 'one': 1, 'six': 6, 'three': 3, 'two': 2}
```

Notice that the key-value pairs are *not* in the order we specified when creating the dictionary. This isn't a problem, though, because we use the keys to look up the corresponding values. We do this using bracket notation, like we did with strings and lists.

```
In [60]: nums['five']

Out[60]: 5
```

If the key does not exist, you'll get an error.

```
In [61]: nums['seven']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)

<ipython-input-61-6d64025bf84a> in <module>()
----> 1 nums['seven']


KeyError: 'seven'
```

We can add the value for 'seven' by doing the following:

```
In [62]: nums['seven'] = 7

In [63]: nums

Out[63]: {'five': 5, 'four': 4, 'one': 1, 'seven': 7, 'six': 6, 'three': 3, 'two': 2}
```

We mentioned earlier that keys can be of almost any type. Values *can* be of any type and we can also mix types.

```
In [64]: mixed = {'one' : 1.0, 'UC Berkeley' : 'Cal', 350 : ['Barrows', 'Hall']}

In [65]: mixed

Out[65]: {'UC Berkeley': 'Cal', 'one': 1.0, 350: ['Barrows', 'Hall']}
```

In this example, we used string and integer keys. We could have actually used any *immutable* objects.

Notice that we used a list as a value, which is valid. What if we tried using a list, which is mutable, as a key?

```
In [66]: {['this'] : 'will not work'}

---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-66-2379a82683dd> in <module>()
----> 1 {['this'] : 'will not work'}


TypeError: unhashable type: 'list'
```

We get a `TypeError` saying that we can't use an unhashable type. What does this mean? In Python, dictionaries are implemented using hash tables. Hash tables use hash functions, which return integers given particular values (keys), to store and look up key-value pairs. For this to work, though, the keys have to be immutable, which means they can't be changed.

Tuples A tuple is a sequence of values. The values, which are indexed by integers, can be of any type. This sounds a lot like lists, right?

> Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain an heterogeneous sequence of elements. . . . Lists are mutable, and their elements are usually homogeneous. . . .

By convention, a tuple's comma-separated values are surrounded by parentheses.

```
In [67]: (1, 2, 3)

Out[67]: (1, 2, 3)
```

Parentheses aren't necessary, though.

```
In [68]: t = 1, 2, 3

In [69]: type(t)

Out[69]: tuple
```

The commas are what define the tuple. In fact, any set of multiple comma-separated objects *without* identifying symbols, such as brackets for lists, default to tuples.

We can't create a tuple with a single element using the following syntax.

```
In [70]: type((1))
```

```
Out[70]: int
```

We need to include a comma following the value.

```
In [71]: type((1,))
```

```
Out[71]: tuple
```

The construction of `t`, above, is an example of *tuple packing*, where the values `1, 2, 3` are "packed" into a tuple.

We can also perform the opposite operation, called *sequence unpacking*.

```
In [72]: a, b, c = t
```

```
In [73]: print(a, b, c)
```

```
1 2 3
```

For this, the number of variables on the left must equal the number of elements in the sequence.

This can be used with functions. In Python, functions can only return a single value. However, that value can be a tuple. In this case, you are effectively returning multiple values.

Most list operators work on tuples. To access tuple elements, for example, we can use the bracket operator.

```
In [74]: t = ('a', 'b', 'c', 'd')
```

```
In [75]: t[0]
```

```
Out[75]: 'a'
```

We can also use the slice operator.

```
In [76]: t[1:3]
```

```
Out[76]: ('b', 'c')
```

Because tuples are immutable, we cannot modify tuple elements.

```
In [77]: t[0] = 'A'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-77-d8f50996a189> in <module>()
----> 1 t[0] = 'A'


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

However, we can create a new tuple using existing tuples.

```
In [78]: t0 = 'A',
         t1 = t[1:]
```

```
In [79]: t0 + t1
```

```
Out[79]: ('A', 'b', 'c', 'd')
```

Sets A set is an unordered collection of unique elements. Because sets are unordered, they do not keep track of element position or order of insertion. As a result, sets do not support indexing or slicing.

> Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

To construct a set, we can use braces ({}) or the built-in `set()` function.

```
In [80]: {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3}
```

```
Out[80]: {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

This returns the *unique* values passed in. In this case, the digits between 1-9, inclusive.
Let's say we had the following list of fruits.

```
In [81]: basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
```

We can find the unique fruits by using the `set()` function.

```
In [82]: set(basket)
```

```
Out[82]: {'apple', 'banana', 'orange', 'pear'}
```

Unlike other built-in Python data structures, sets support differencing.

```
In [83]: {1, 2, 3} - {2}
```

```
Out[83]: {1, 3}
```

```
In [84]: {1, 2, 3} - {1, 2, 3}
```

```
Out[84]: set()
```

Sets are useful for finding unique values and for performing mathematical operations like the ones previously mentioned.

Python also provides "specialized" container types in its `collections` module. These are alternatives or, rather, complements, to Python's general-purpose, built-in containers that we've just coveredlists, dictionaries, tuples, and sets. For more information on these other data structures, see the documentation.

In the following section, we'll explore several operators that the data structures covered above respond to.

1.3.2 Operators

There are several operators supported in Python. They are:

- arithmetic
- comparison (relational)
- assignment
- logical
- bitwise
- membership
- identity

We've already covered some of these either directly or in passing. We'll discuss how some of these operate on the data structures we've learned about thus far.

Arithmetic The arithmetic operators are the ones you're probably most familiar with. These include `+`, `-`, `*`, `/`, and `**` to name a few. Of course, not all of these work on all Python data types.

Previously, we saw how the `+` and `*` operators, which correspond to concatenation and repetition, operate on strings. It turns out that lists and tuples respond in similar ways.

```
In [85]: [1, 2, 3] + [4, 5, 6]

Out[85]: [1, 2, 3, 4, 5, 6]

In [86]: (1, 2, 3) + (4, 5, 6)

Out[86]: (1, 2, 3, 4, 5, 6)

In [87]: ['Cal'] * 3

Out[87]: ['Cal', 'Cal', 'Cal']

In [88]: ('D-Lab',) * 3

Out[88]: ('D-Lab', 'D-Lab', 'D-Lab')
```

Comparison These type of operators "compare the values on either sides of them and decide the relation among them."

```
In [89]: [1, 2, 3] == [1, 2, 3]

Out[89]: True

In [90]: [0, 2, 3] == [1, 2, 3]

Out[90]: False
```

> The comparison uses *lexicographical* ordering: first the first two items are compared, and **if they differ this determines the outcome of the comparison**; if they are equal, the next two items are compared, and so on, until either sequence is exhausted.

```
In [91]: [0, 2, 3] < [1, 2, 3]

Out[91]: True
```

In the comparison above, because the `0` is less than the `1`, the result is `True`. Once this is determined, subsequent values are *not* compared. In the example below, the return value is `True` even though `20` is greater than `2`.

```
In [92]: [0, 20, 30] < [1, 2, 3]

Out[92]: True
```

The behavior is the same with tuples.

```
In [93]: (0, 20, 30) < (1, 2, 3)

Out[93]: True

In [94]: (0, 1, 2) == (0, 1, 3)

Out[94]: False
```

Interestingly, the behavior is slightly different with sets. Consider the list and set comparisons below.

```
In [95]: [0, 3, 4] < [1, 2, 9]

Out[95]: True

In [96]: set([0, 3, 4]) < set([1, 2, 9])

Out[96]: False
```

With sets, the comparisons are made for every element in each corresponding sequence. Comparisons can be made with dictionaries, too.

```
In [97]: {'one' : 1} == {'one' : 1}

Out[97]: True
```

But we can only check for equality.

```
In [98]: {'one' : 1} < {'one' : 1}


---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-98-da991da360aa> in <module>()
----> 1 {'one' : 1} < {'one' : 1}


TypeError: unorderable types: dict() < dict()
```

Membership These operators test for membership—that is, whether the particular item exists—in a sequence.

```
In [99]: 'D-Lab' in ['D-Lab', 'UC Berkeley']

Out[99]: True

In [100]: 1 in (0, 1, 2)

Out[100]: True

In [101]: 99 in {1868, 350, 102}

Out[101]: False
```

For dictionaries, membership is tested against the keys.

```
In [102]: cities = {'Berkeley' : 'California',
                    'Miami' : 'Florida',
                    'New York' : 'New York',
                    'Seattle' : 'Washington'}

In [103]: 'Berkeley' in cities

Out[103]: True
```

The other membership operator is `not in`.

```
In [104]: 99 not in {1868, 350, 102}

Out[104]: True
```

Identity To compare the memory locations of objects, use identity operators.

```
In [105]: a = 'this'
          b = 'this'

In [106]: a is b

Out[106]: True
```

In this case, the memory address for both `a` and `b` is the same because they are pointing to the same value.

Is this behavior consistent across daya types?

```
In [107]: a = 1868
          b = 1868

In [108]: a is b

Out[108]: False
```

Apparently not.

```
In [109]: hex(id(a)), hex(id(b))

Out[109]: ('0x10428b6d0', '0x10428b890')
```

What if we set `b` to equal `a`?

```
In [110]: b = a

In [111]: a is b

Out[111]: True
```

Like with the membership operator, the complement to `is` is `is not`.

```
In [112]: a = 'this'
          b = 'that'

In [113]: a is not b

Out[113]: True
```

1.3.3 Functions and Methods

We're familiar with functions, but what are methods?

> A method is a function that "belongs to" an object.

We have already seen string methods. For example, from day zero:

```
>>> my_string = 'Dav Clark wears a beret'
>>> my_string = my_string.replace('beret', 'speedo')
>>> print(my_string)
Dav Clark wears a speedo
```

Here, `.replace()` is the method.

Python data structures have methods, too.

Lists Let's use `first_list`, which we created above, to demonstrate some list functions and methods.

Let's say we wanted to know how many elements are in `first_list`. For this, we would use the `len()` function.

```
In [114]: len(first_list)
```

```
Out[114]: 5
```

What about the largest and smallest values

```
In [115]: max(first_list), min(first_list)
```

```
Out[115]: (9, 5.43)
```

Let's say we wanted to add an element to `first_list`. For this, we can use the `.append()` method.

```
In [116]: first_list.append(2)
```

Notice that methods are called using dot notation on the object we'd like to modify.

By default, `.append()` adds an element to the *end* of a given list.

```
In [117]: first_list
```

```
Out[117]: [9, 8, 7.0, 6, 5.43, 2]
```

Notice how we invoked this method. We did not use an assignment operator (e.g., `x = x.append(y)`). This is because—and this is important—list methods are all void, which means that they *modify* lists and return `None`.

Sometimes when we're adding elements to a list, we may with to insert it in a given position. For this, we can use the `.insert()` method. It takes two arguments—the first is the *position* and the second is the *value*. Let's say we wanted to add an item to the front of the list. We could do it using:

```
In [118]: first_list.insert(0, 10)
```

```
In [119]: first_list
```

```
Out[119]: [10, 9, 8, 7.0, 6, 5.43, 2]
```

Let's append another value to the list.

```
In [120]: first_list.append(2)
```

Now, let's count how many times the value `2` appears.

```
In [121]: first_list.count(2)
```

```
Out[121]: 2
```

```
In [122]: first_list
```

```
Out[122]: [10, 9, 8, 7.0, 6, 5.43, 2, 2]
```

Let's say we wanted to remove one of the 2s.

```
In [123]: first_list.remove(2)
```

```
In [124]: first_list
```

```
Out[124]: [10, 9, 8, 7.0, 6, 5.43, 2]
```

The remove method removes the *first* item in the list that matches the value in the parentheses.

In some cases, we might want to know the index value for a certain list element. We can use `.index()` for this.

```
In [125]: first_list.index(5.43)
```

```
Out[125]: 5
```

The value `5.43` can be found at index `5`.

More information on list methods can be found here.

Dictionaries Let's use our `nums` dictionary to demonstrate some `dict` methods.

```
In [126]: nums

Out[126]: {'five': 5, 'four': 4, 'one': 1, 'seven': 7, 'six': 6, 'three': 3, 'two': 2}
```

The `len()` function we saw above also works on dictionaries. It returns the number of items in the object.

```
In [127]: len(nums)

Out[127]: 7
```

We might be interested in getting a list of the keys in `nums`. The `.keys()` method returns a list with this information.

```
In [128]: nums.keys()

Out[128]: dict_keys(['two', 'three', 'four', 'six', 'five', 'seven', 'one'])
```

We can do the same for values.

```
In [129]: nums.values()

Out[129]: dict_values([2, 3, 4, 6, 5, 7, 1])
```

To add to the dictionary, we can use the `.update()` method.

```
In [130]: nums.update(eight=8)

In [131]: nums

Out[131]: {'eight': 8,
           'five': 5,
           'four': 4,
           'one': 1,
           'seven': 7,
           'six': 6,
           'three': 3,
           'two': 2}
```

Notice that we don't use quotation marks around the key name `eight`.

If we'd like to remove an item, we can use the `.pop()` method. This removes the item—the key-value pair—*and* returns the value.

```
In [132]: nums.pop('one')

Out[132]: 1

In [133]: nums

Out[133]: {'eight': 8, 'five': 5, 'four': 4, 'seven': 7, 'six': 6, 'three': 3, 'two': 2}
```

We've successfully removed {`one : 1`} from `nums`.

Tuples Tuples have no methods.

Sets There are several set methods. They can be used for updating set objects or for performing mathematical operations. For example, we can add an element to set `s`.

```
In [134]: s = {1, 8, 6, 8}

In [135]: s

Out[135]: {1, 6, 8}

In [136]: s.add(0)

In [137]: s

Out[137]: {0, 1, 6, 8}
```

We can also remove set elements.

```
In [138]: s.remove(1)

In [139]: s

Out[139]: {0, 6, 8}
```

Python supports several mathematical operations on sets. We can check the intersection—or overlap—of two sets, for example.

```
In [140]: {1, 2, 3} & {3, 4, 5} # or {1, 2, 3}.intersection({3, 4, 5})

Out[140]: {3}
```

Another common set operation is the union, which basically combines sets.

```
In [141]: {0, 1} | {1, 2} # or {0, 1}.union({1, 2})

Out[141]: {0, 1, 2}
```

Above, we saw that `{1, 2, 3} - {2}` resulted in `{1, 3}`. However, if the second set had more values in it, those values would not be represented in the final set. Python sets allow you to calculate the symmetric difference:

```
In [142]: {1, 2, 3} ^ {3, 4, 5}

Out[142]: {1, 2, 4, 5}
```

Along with testing for supersets and subsets

```
In [143]: {1, 2, 3} > {2, }

Out[143]: True
```

Data structures **DOCS**

1.4 Control Flow

DOCS

1.4.1 `for`

In Python, a `for` statement iterates over items in a sequence—such as strings, lists, and tuples—in the order that they appear. `for` loops have the following syntax.

```
for item in sequence:
    do_something_with(item)
```

> side note - for whatever reason, some students have a really hard time with for loop syntax. Emphasize that in `for x in sequence`, `x` is an arbitrary name so that you can refer to the object returned by the iterator while you are inside of the loop. You could also use `for dinosaur in sequence`, but this reduces readibility in your code

The `sequence` object should be iterable. The `statement(s)` are executed once for each item in the sequence. This is referred to as traversing the sequence. The loop ends when there are no more elements in the sequence.

Let's look at some examples.

```
In [144]: text_var = 'berkeley'

In [145]: for c in text_var:
              print(c)

b
e
r
k
e
l
e
y
```

With strings, the `for` statement iterates over each character. With lists (or tuples), on the other hand, each list element is iterated over.

```
In [146]: list_var = [350, 'Barrows', 'Hall']

In [147]: for e in list_var:
              print(e)

350
Barrows
Hall
```

With dictionaries, `for` loops iterate over keys.

```
In [148]: for k in {'one' : 1, 'two' : 2, 'three' : 3}:
              print(k, end=" ")

three one two
```

If we'd like a loop that iterates a given number of times or over a sequence of numbers, we can use the `range` object.

```
In [149]: for v in range(4):
              print(v, end=" ")

0 1 2 3
```

1.4.2 `while`

Another way to achieve this—to iterate a given number of times—is to use the `while` loop.

```
In [150]: n = 0
          while n < 4:
              print(n, end=" ")
              n += 1
          print('\ndone')
```

```
0 1 2 3
done
```

In this example, we have to increment `n` with each iteration of the loop. The body statements in `while` loops repeatedly execute as long as the header condition evaluates to `True`. Once the loop ends, program control passes to the line immediately following the loop.

With `while` loops, there are two possibilities to be aware of. First, it's possible that some `while` loops never execute. Using the code above, if the value of `n` is initially `4` or greater, only `done` will be printed.

```
In [151]: n = 4
          while n < 4:
              print(n, end=" ")
              n += 1
          print('\ndone')
```

```
done
```

Above, because the condition evaluates to `False`, the loop body is skipped and the first statement after the `while` loop is executed.

Second, some `while` loops may run indefinitely. This is referred to as an infinite loop and happens when the condition *never* evaluates to `False`. Here is an example.

```
n = 4
while n >= 4:
    print(n, end=" ")
    n += 1
print('\ndone')
```

1.4.3 `if`

In many cases, it's useful to control the order in which statements or function calls are executed or evaluated. A control flow statement determines which path or paths in a program should be followed. Control flow statements, for example, can:

- execute a set of statements if a condition or certain conditions are met
- execute a set of statements `n` times until a condition or certain conditions are met
- stop the execution of a program

How can we achieve this? The most well-known statement type is the `if` statement.

```
In [152]: x = 0
```

```
In [153]: if x == 0:
              print('x is zero')
```

```
x is zero
```

`if` statements make use of boolean expressions. If the expression (or set of expressions) evaluate to `True`, the indented statement gets executed. Otherwise, nothing happens.

```
In [154]: x = 1

In [155]: if x == 0:
              print('x is zero')
```

The code above is referred to as a clause. Clauses contain "headers" and "bodies." Clause headers begin with identifying keywords—in this case, `if`—include boolean expressions, and end with colons. The body is a group of indented statements controlled by the clause. This is also known as a "block."

Compound statements are made up of one or more clauses. For example, there might be two possibilities in which case we use the `else` keyword. We can combine the above as follows.

```
In [156]: if x == 0:
              print('x is zero')
          else:
              print('x is not zero')
```

```
x is not zero
```

Notice that clause headers are at the same indentation level.

When there are more than two possibilities, we can use what are called chained conditionals. For this, we use the `elif` keyword.

```
In [157]: if x == 0:
              print('x is zero')
          elif x < 0:
              print('x is negative')
          elif x > 0:
              print('x is positive')
```

```
x is positive
```

Of course, the code above only works if `x` is numeric. Assuming this is the case, all possible values of `x` are listed. Because of this, we can change the last clause (`elif x > 0`) to `else`.

There isn't a "right" way to do this. A good approach is to write it such that its easily readable for yourself and others.

What if `x` is *not* numeric? With the code as is, we'll get a `TypeError`. So, let's generalize what we have and wrap it in a function.

```
In [158]: def x_is(x):
              if type(x) is str:
                  print('x is str')
              elif type(x) in [int, float]:
                  if x == 0:
                      print('x is zero')
                  elif x < 0:
                      print('x is negative')
                  elif x > 0:
                      print('x is positive')
              else:
                  print('invalid x value')
```

Before we call our function, let's explain what's going on. Our function, as defined, is an example of a "nested conditional." We first perform a type check and, if `x` is numeric, there are another set of conditions which are checked.

```
In [159]: x_is('ucb')

x is str

In [160]: x_is(1)

x is positive

In [161]: x_is(0)

x is zero

In [162]: x_is([1, 2, 3])

invalid x value

In [163]: x_is(None)

invalid x value
```

Control flow **DOCS**

1.5 Input and Output

DOCS

Interacting with data in files is a common task in Python. These can be plain text files, comma-delimited (CSV) files, or any other number of file formats.

To open files, we can use the built-in `open()` function. There is a file named `lorem-ipsum.txt` in the `data/` directory that we'll use to learn about file input and output.

The `open()` function is typically used with two arguments—the filename and the "mode." The mode describes how the file will be used. The default is `r`, which stands for "read only."

```
In [164]: f = open('../data/01_lorem-ipsum.txt', 'r')
```

`f` is a file object. There are several methods we can use to interact with the file's contents.

The `.read(size)` method reads the contents of the file object. The optional numeric argument, ***size***, corresponds to the number of bytes that should be read. This is useful if the data file is large. If we omit `size`, the entire contents of the file will be read and returned.

```
In [165]: f.read()

Out[165]: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor\nincididunt u
```

What happens if we try to call `f.read()` again? "If the end of the file has been reached, `f.read()` will return an empty string ()." In this situation, the "cursor" is at the end of the file and has nothing more to read.

Because we'd like to show a few other methods, we can return to the beginning of the file using the `.seek()` method, passing in `0` as the argument.

```
In [166]: f.seek(0)

Out[166]: 0
```

Let's say we wanted to read the file, line-by-line. We can accomplish this using the `.readline()` method. The end of a "line" is identified by the presence of a new line character, `\n`. You can see some in the text output above.

```
In [167]: f.readline()
```

```
Out[167]: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor\n'

In [168]: f.readline()

Out[168]: 'incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis\n'
```

And so on.
If you ever need to know the file object's current position, use the `.tell()` method.

```
In [169]: f.tell()

Out[169]: 154
```

This represents the number of *bytes* from the beginning of the file.
We can also loop over the file object. Let's return to the start of the file first.

```
In [170]: f.seek(0)

Out[170]: 0

In [171]: for line in f:
              print(line)

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor

incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis

nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu

fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in

culpa qui officia deserunt mollit anim id est laborum.
```

When we're done interacting with a file, that fie should always be closed.

```
In [172]: f.close()
```

We can always check whether a file is closed by using the following.

```
In [173]: f.closed

Out[173]: True
```

The `with` keyword in Python ensures that files are properly closed after its associated code is executed. This is true even if an exception is raised. Using the `with` keyword is recommended.
Let's print each line on our document using this syntax.

```
In [174]: with open('../data/01_lorem-ipsum.txt', 'r') as f:
              for line in f:
                  print(line)

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor

incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis

nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
```

```
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu

fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in

culpa qui officia deserunt mollit anim id est laborum.
```

We can also check that the file was, indeed, closed.

```
In [175]: f.closed

Out[175]: True
```

What about writing to a file? There are two primary modes we can use for this: `w` for writing only and `a` for appending to a file. If a file opened in `w` mode already exists, it will be overwritten. Opening a file in `a` mode simply allows lines to be added to the end of an existing file.

Let's start by creating a new file.

```
In [176]: with open('first-write.txt', 'w') as f:
              f.write('this is our first line\n')
              f.write('this is our last line')
```

Now, let's check the contents of the file.

```
In [177]: with open('first-write.txt', 'r') as f:
              for line in f:
                  print(line)

this is our first line

this is our last line
```

Note that while we've been using `f` to identify our file object, we can use any valid variable name.

Now, let's append to our file.

```
In [178]: with open('first-write.txt', 'a') as append_file:
              append_file.write('\nthis is the real last line')
```

Notice that we add a new line character to the beginning of this third line.

```
In [179]: with open('first-write.txt') as infile:
              for row in infile:
                  print(row)

this is our first line

this is our last line

this is the real last line
```

In the code above, we use `row` where we had previously used `line`. We did that to serve as a reminder that the variable names used are not special in any way. It is, however, always a good idea to use descriptive variable names that make reading the code more understandable. This is part of making code "readable." For a bit more on this, see here, here, and here.

The `open()` function can take a variety of file types. We've seen examples of how to use this with a `.txt` file.

The CSV (comma separated values) format is "the most common import and export format for spreadsheets and databases."

> [A] comma-separated values (CSV) file stores tabular data (numbers and text) in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas.

We can open comma-delimited CSV files with `open()`, too. Let's open an example CSV file in `data/` called `roster.csv`.

```
In [180]: with open('../data/01_roster.csv', 'r') as roster:
              for student_data in roster:
                  print(student_data)

id,department,level,age,sex

8349,sociology,graduate,28,m

4922,economics,undergraduate,20,m

5091,physics,graduate,23,f

1157,linguistics,undergraduate,22,f

2109,"environmental science, policy, and management",undergraduate,19,m

7307,engineering,graduate,30,f
```

This file includes some made-up student information—a four-digit ID number, academic status, and demographic data.

In some cases—say, if we need to calculate the average age of these students—we don't actually want to iterate over the first row, which is often called the "header."

```
In [181]: with open('../data/01_roster.csv', 'r') as roster:
              next(roster)
              for student_data in roster:
                  print(student_data)

8349,sociology,graduate,28,m

4922,economics,undergraduate,20,m

5091,physics,graduate,23,f

1157,linguistics,undergraduate,22,f

2109,"environmental science, policy, and management",undergraduate,19,m

7307,engineering,graduate,30,f
```

We do this using the `next()` function, which just goes to the next line. In this case, since we're starting at the top of the file, it goes to the second line.

Now, let's say we wanted to create a list of the six student ages. How might we go about doing that? One approach might be to split each line on commas to extract the age. This would work expect for the fact that student `2109`'s department *includes* a comma in the value.

To help with situations like these, Python has a built-in `csv` module which includes lots of functionality for working with these types of types. Let's show how we could use this to calculate the average age of the students.

```
In [182]: import csv

In [183]: ages = []
          with open('../data/01_roster.csv', 'r') as f:
              next(f)
              roster = csv.reader(f, delimiter=',', quotechar='"')
              for student_data in roster:
                  ages.append(int(student_data[3]))
```

The `reader()` function allows us to specify the delimiter and the quote character. The quote character, in this case, is the quotation mark ("). CSV files often wrap string values in quotes (or other characters) if they include the delimiter within them. The `reader()` function parses each line as a list of strings, taking into consideration the delimiter and quote character. This is why we can select the third element in `student_data` and why we change (or cast) the type to `int`. As we iterate over each line, we add the age value to `ages`.

Now, we can create a new variable that holds the ages and calculate the average.

```
In [184]: ages_mean = sum(ages) / len(ages)

In [185]: print('The average age of students in the roster is: %.2f' % ages_mean)

The average age of students in the roster is: 23.67
```

The `%.2f % ages_mean` simply instructs Python to print the value in `ages_mean` to two decimal places.

Input output **DOCS**

1.6 os

DOCS

It is often useful and sometimes necessary to interact with the operating system. For example, we might be interested in modifying file paths or getting a list of files in a given directory. Python's built-in `os` module provides "operating system dependent functionality."

To start, let's import `os`.

```
In [186]: import os
```

Let's begin by listing our current working directory.

```
In [187]: os.getcwd()

Out[187]: '/Users/dillon/Dropbox/dlab/workshops/python-for-everything/instructor'
```

We know we have a `data/` directory in our repository, but we might not know its contents. We can get that information by using the following.

```
In [188]: os.listdir('../data/')

Out[188]: ['01_lorem-ipsum.txt',
           '01_roster.csv',
           '02_tweet.json',
           '03_cities.csv',
           '03_feedback.csv',
           '03_people.csv',
           '03_text.md',
           '2015-01-04-carto-export.csv']
```

This results in a list of the entries in the directory (excluding `.` and `..`). Notice that we're able to specify a *relative* path with `listdir()`.

If we were writing a Python script that used one of these files, we might want to include checks for whether or not the files exist. We can also accomplish this with `os`. First, we can check if a directory exists.

```
In [189]: os.path.isdir('../data/')
```

```
Out[189]: True
```

We can also check to see if a file exists.

```
In [190]: os.path.isfile('../data/01_roster.csv')
```

```
Out[190]: True
```

Both of these return a Boolean value. One way these could be used is in conjunction with `if` statements. An alternative, the `os.path.exists()` function, checks for either files or directories.

If a directory doesn't exist, we can create it from within Python. This is accomplished using the `mkdir()` function, which takes a file path as an argument.

```
In [191]: os.mkdir('newdir')
```

Let's check the contents of the current directory.

```
In [192]: os.listdir()
```

```
Out[192]: ['.ipynb_checkpoints',
           'day_one.ipynb',
           'day_one.pdf',
           'day_one.py',
           'day_three.ipynb',
           'day_three.pdf',
           'day_three.py',
           'day_two.ipynb',
           'day_two.pdf',
           'day_two.py',
           'day_zero.ipynb',
           'day_zero.pdf',
           'day_zero.py',
           'first-write.txt',
           'images',
           'newdir']
```

We can use the `rmdir()` function to remove `newdir/`.

```
In [193]: os.rmdir('newdir')
```

For more information on the available functions, see the documentation.

`os` **DOCS**

1.7 `glob`

DOCS

It's sometimes necessary to find file or pathnames matching a particular pattern. Python's built-in `glob` module uses Unix shell-style wildcards for pattern matching. Note that these are different from regular expressions.

There is no shell variable (e.g., `$PATH`) or tilde (`~`, which typically refers to the "home" directory) expansion in `glob`. In addition, `glob` does not show hidden files—those that start with dots (`.`).

Below we describe the behavior of the shell-style wildcards.

Pattern	Meaning
*	matches everything
?	matches any single character
[seq]	matches any character in seq
[!seq]	matches any character not in seq

Above, when we used `os.listdir()` in our current directory, the returned list included the Jupyter notebook files as well as a directory and the `.ipynb_checkpoints` file. Let's see what `glob` returns.

```
In [194]: import glob

In [195]: glob.glob('*')

Out[195]: ['day_one.ipynb',
           'day_one.pdf',
           'day_one.py',
           'day_three.ipynb',
           'day_three.pdf',
           'day_three.py',
           'day_two.ipynb',
           'day_two.pdf',
           'day_two.py',
           'day_zero.ipynb',
           'day_zero.pdf',
           'day_zero.py',
           'first-write.txt',
           'images']
```

Notice that the list does not include `.ipynb_checkpoints`.
Let's use `glob` to show only the `.ipynb` files.

```
In [196]: glob.glob('*.ipynb')

Out[196]: ['day_one.ipynb', 'day_three.ipynb', 'day_two.ipynb', 'day_zero.ipynb']
```

If we want directories only.

```
In [197]: glob.glob('*/')

Out[197]: ['images/']
```

The `*` matches zero or more characters.
Let's create a few directories (and a file) to make this concrete.

```
In [198]: !mkdir test
          !mkdir test1
          !mkdir test10
          !mkdir test100
          !touch test.txt
```

Note that the `!` before each line above allows us to run shell commands from within the notebook.

```
In [199]: glob.glob('test*')

Out[199]: ['test', 'test.txt', 'test1', 'test10', 'test100']
```

This returns any file or directory that begins with `test` and end with any (or no other) character.

We can also match directories only.

```
In [200]: glob.glob('test*/')

Out[200]: ['test/', 'test1/', 'test10/', 'test100/']
```

To match a single character, we can use the `?` wildcard character. This matches any character in the specified position of the name.

```
In [201]: glob.glob('test?')

Out[201]: ['test1']
```

In this case, the only match is `test1`, which we know is a directory.

Next, let's show what the character range (`[]`) wildcard can do. We'll create a few more directories (we'll clean this up when we're done).

```
In [202]: !mkdir tset0
          !mkdir tset1
          !mkdir tset5
          !mkdir tset10
          !mkdir tset50
```

The character range wildcard matches a single character in the specified range.

```
In [203]: glob.glob('tset[0-1]')

Out[203]: ['tset0', 'tset1']
```

The code above matches files or directories that start with `tset` and that end with either `0` or `1`. If we were to have used `0-9` in the brackets, it would have also returned `tset5`.

If we want the directories that end with *two* digits, we can do the following.

```
In [204]: glob.glob('tset[0-9][0-9]')

Out[204]: ['tset10', 'tset50']
```

The character range wildcard also works on letters.

```
In [205]: glob.glob('t[a-z][a-z]t?')

Out[205]: ['test1', 'tset0', 'tset1', 'tset5']
```

This matches files or directories that begin with a `t` and are followed by two letters, a `t`, and a single character.

An alternative way of getting the same result is as follows.

```
In [206]: glob.glob('t??t?')

Out[206]: ['test1', 'tset0', 'tset1', 'tset5']
```

This is because we don't have any files or directories with numbers in the second and third positions.

Let's clean up our directory.

```
In [207]: !rm -rf test*
          !rm -rf tset*
```

`glob` **DOCS**

1.8 subprocess

DOCS

> A running program is called a **process**.

It contains code and its associated activity (or state). For example, this includes memory, lists of open files, etc.

Programs, which are processes, can also create new processes. These are known as **subprocesses** and independently of the processes which created (or spawned) them. This means this new process can run at the same time as the original.

Python's `subprocess` module provides an interface for creating and working with additional processes.

When might we want to spawn new processes? One example is executing a Python script—much like you would from the command line—within Python. Although we know that we can use the `!` to run shell commands, this only works from within the notebook. So, let's use `subprocess` to execute a Python script in `scripts/` named `simple.py`.

```
In [208]: import subprocess
```

```
In [209]: subprocess.check_output(['python', '../scripts/simple.py'])
```

```
Out[209]: b'IOKN2K!\n'
```

This file print's `IOKN2K!` (and a new line character, `\n`), which is an abbreviation for, "it's okay not to know!"

With `check_output()`, the command to be executed must be passed in as a list. Each argument of the command should be a separate list element. `check_output()` lets us execute an external command and collect its output.

The `b` prefex indicates that the returned value is a bytes type as opposed to a `str` type. If needed, we can convert this using the following.

```
In [210]: subprocess.check_output(['python', '../scripts/simple.py']).decode('utf-8')
```

```
Out[210]: 'IOKN2K!\n'
```

`subprocess` **DOCS**