Manipulating Variables in Python

* Core Python: Small Yet Powerful
  + While working in Python, you'll only use a small set of components at any given time. These are:
    - Keywords.
    - Built-in functions.
    - Built-in data types (and associated methods).
    - Explicitly imported names from modules.
  + Programming is the art of using this small set of components to express thoughts. And, using only Python's built-in data types, keywords, and functions, you can actually implement any algorithm that operates on in-memory data.
  + Let's take a close look at each component in the set.
* Keywords
  + Keywords are specific, reserved words that have special meaning to Python.
    - 
  + You might recognize some of these, or can at least guess what they do.
  + For now, just keep in mind that:
    - There are very few keywords, and they are not complicated
    - Keywords cannot be used as variable names
    - Using these keywords might require special syntax. For example, if must be followed by a valid expression, which must be followed by a colon
* Built-In Functions
  + Python comes with a small-yet-powerful set of built-in functions. Surprisingly, all you need to implement any algorithm are these functions (along with keywords and built-in data types).
  + Keep in mind that:
    - There are not many built-in functions, and you can easily guess what most accomplish.
    - Each built-in function is simply a name that points to a function object.
    - We've already seen a lot of these — e.g., dir, type, and the constructor function for each data type (int, float, etc.).
  + We already discussed data types (no. 3), so now we'll move on to modules.
* Modules
  + In Python, you can't use an undefined name without defining or importing it first. For example, you can't call the math module's sin function without importing it.
  + Imports typically go at the very top of your script or Jupyter Notebook. If you see an unfamiliar name that isn't built in, then it's nearly always the result of an import at the top of the file. There are three typical ways to import in Python. An easy way to remember them is that, whatever comes after the keyword import is the name that will be defined in your code.
    - import math: This imports everything in the math module under the name math. To access any math functions, you would call them using the method notation (e.g., math.sin(0)).
    - from math import sin: This imports only the sin function from the math module. To use the sin function, simply call sin(0).
    - from math import \*. This imports every function in the math module as its own name. This is considered bad practice, because it's unclear which names have been added by just looking at the import statement. It would be easy to accidentally overwrite one of these imported names.
* Lists
  + Now, let's take a look at the built-in containers. A container is simply a data type that can hold more than one object.
  + Suppose we have three types of animals. One way of storing the animal types is by using three separate variables:
    - animal\_type1 = 'bird'
    - animal\_type2 = 'giraffe'
    - animal\_type3 = 'monkey'
  + As you might have guessed, this gets to be cumbersome. Down the line we'll have difficulty storing more animal types and looking through all of the types we already have.
  + Thankfully, we have lists!
    - animal\_types = ['bird', 'giraffe', 'monkey']
    - As expected, type(animal\_types) evaluates to list.
  + Here are some important points to remember about lists:
    - Lists are ordered. Their elements have a particular order that will never change.
    - Lists are heterogeneous. Different data types can be stored for each element in the list. For example, ['cat', 10, 0.4].
    - Lists are mutable. When you alter a list, you don't create a new object — the original object is just modified.
  + Typically, the name of a list should be plural. For example, cars, animal\_names, and cities. This immediately indicates that the name refers to a container.
  + animal\_types = ['bird', 'giraffe', 'monkey']
    - Recall that len() is a built-in function that takes any container as a parameter. So, len(animal\_types) is 3.
    - Similar to strings, we can use the indexing operator ([]) to access any particular element in the list. So, animal\_types[0] is 'bird'.
    - Unlike strings, we can set any element to a new value. For example, animal\_types[0] = 'cat' replaces 'bird' with 'cat' as the first element.
* Literals
  + In general, a literal is a text representation of a particular data type. When Python interprets the text, this representation is converted into the 1s and 0s of the actual object. For example, 'I live in a city.' is a string literal. However, I live in a city does not represent a string — Python looks up the individual names to find the objectives to which they are referring.
  + Note that brackets have two meanings in Python:
    - If a bracket does not immediately follow an object or name, it represents a list. This is called a list literal — a text representation of a list. For example, days = ['Monday', 'Tuesday'].
    - If a bracket immediately follows an object or name, it indexes into the object. For example, days[0] indexes into the list referenced by the name days.
  + In print(['Monday', 'Tuesday'][0]) we find both uses of a bracket — as a literal AND an index. Because the list literal is converted to an object, the second use of brackets indexes into that object.
* List Methods
  + Like before, you can use the built-in dir() function to view the methods available to a list object (dir(list)). Alternatively, if you're using ipython, press tab after typing a period following a list object. The most important list method is append(), which adds a new element to the end of a list. For example:
    - animal\_types = ['bird', 'giraffe', 'monkey']
    - animal\_types.append('turkey')
  + After evaluating both lines, animal\_types contains four elements: ['bird', 'giraffe', 'monkey', 'turkey'].
  + Another important list method is .pop() which removes an element from a list. If you encounter a method you haven't seen before, or don't know the name of a method for a task you want to accomplish, you can Google it (no joke! Google can be your best coding buddy, and it's both valuable and rewarding to practice finding things you don't know or remember).
  + Unlike string methods, these alter the original list. To view the altered list in the REPL, simply evaluate its name after running the method (e.g., animal\_types).
  + Why is that? Because the primitive data types in the last lesson — Booleans, numbers, strings — are immutable, and an immutable object never changes its value. Instead, the interpreter creates a new object and directs the name to it.
  + For example, evaluating 312 + 457 actually creates a new int object, 769. On the other hand, the methods of most containers (excluding strings and tuples) alter the same object for efficiency's sake. The language designers decided to not return a list to emphasize to the programmer that lists are mutable — the original was changed. This prevents you from making common mistakes.
* Tuleps
  + Tuples are similar to lists in that they are containers of objects. However, there are two main differences:
    - They are defined using parentheses instead of brackets.
    - They are immutable — you cannot alter a particular tuple object once it is created. So, most methods such as append() and pop() do not exist.
  + You can define a new tuple like so:
    - # (month number, month name, num days)
    - month = (1, 'January', 31)
* Lists vs Tuleps
  + Use lists when order is significant and elements are of the same data type.
  + Use tuples when order is not significant and the elements collectively represent something.
  + For example, suppose we have a computer defined by its name, amount of RAM, and date of manufacture.
    - my\_computer = ('WORK\_COMPUTER', 8, 2017)
  + Here, it is not meaningful that the computer name comes before the RAM amount. The order could have just as easily been defined differently without consequences. Additionally, the elements have mixed data types that are attributes of a whole. Therefore, this computer is best represented as a tuple.
  + Next, suppose we have a list ordered by most-used computer.
    - most\_used\_computers = ['WORK\_COMPUTER', 'HOME\_COMPUTER', 'LAPTOP']
    - Because these are all strings (homogeneous) and order matters, this is best represented as a list. We also might want to add additional computers to this list in the future.
  + You may be wondering why we would use tuples if they are actually more restrictive than lists. Making a data type immutable allows it to be more efficiently represented in memory. Furthermore, immutable types can be used in some additional objects (such as keys to dictionaries).
* Sets
  + A set is a collection of unordered, unique elements. For example, two people may have watched different sets of movies.
    - my\_movies\_watched = {'Finding Dory', 'Dumbo', 'Beauty and the Beast'}
    - your\_movies\_watched = {'Wizard of Oz', 'Finding Dory'}
  + In this case, the particular order inside the set does not matter, as each movie was either watched or it wasn't. Note that Python knows these are sets because the strings are surrounded by curly braces and only contain a simple container of elements, which are separated by commas.
  + Pro tip: To create an empty set, use the built-in set() function. For example, untasty\_fruits = set(). Why? Because to Python, curly braces are used for both sets and dictionaries. Empty braces {} indicate an empty dictionary.
  + Important points to remember about sets:
    - Sets are unordered. Never use a set if order matters.
    - Sets contain no duplicate elements. {1, 2, 2} is equivalent to {1, 2}.
    - Membership look ups are fast. Determining whether or not an element is included in a set often takes a constant amount of time.
  + To find out whether or not a set contains an element, use the in operator: 'Finding Dory' in my\_movies\_watched evaluates to True.
* Dictionaries
  + Dictionaries represent key-value pairs. This just means a term can be stored with specific attributes. For example:
    - Names could be paired with ages.
    - Months could be paired with the number of days in the month.
    - Book titles could be paired with authors.
  + As with sets, a dictionary is defined using curly braces. However, each element of a set consists of a key, followed by a colon, followed by a value. The colons distinguish between a set literal and a dictionary literal. For example:
    - book\_authors = {'Moby Dick': 'Herman Melville', 'The Lorax': 'Dr. Seuss', 'The Hobbit': 'J.R.R. Tolkien'}
  + Notice this is a container of key-value pairs. So, len(book\_authors) is 3, as there are three key-value pairs.
  + Alternatively, you can think of a dictionary as a generalized list. Instead of indexing items by integers, we now index them by their keys. For example, book\_authors['Moby Dick'] evaluates to 'Herman Melville'.
  + The keys can be any immutable data type, while the values can be any data type. For instance:
    - points\_grades = {0: ['Fail', 'F'], 10: ['Perfect', 'A+']}
    - Then, a student receiving 10 points would get the grade points\_grades[10], which would evaluate to the list ['Perfect', 'A+'].
  + To add a new key-value pair to the dictionary, index into a new key: book\_authors['Origin'] = 'Dan Brown'. This adds a new key, 'Origin', with a value of 'Dan Brown'.
  + Another way of thinking about a dictionary is as a set of keys and a list of values. The container of keys must be a set, because each key is unique (i.e., has at most one value). With this in mind, two useful dictionary methods are:
    - book\_authors.keys()
    - book\_authors.values()
  + Similar to a set, assume that a dictionary is unordered and that the key-value pairs do not exist in any particular order.
* Container Data Types

