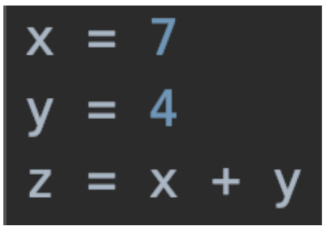
The Python language

Coding languages are not much different than spoken/written languages. There are rules for written languages:

- Punctuation (e.g. periods, commas, etc.) carries meaning. For example, a period indicates the end of a complete thought.
- Complete sentences must follow a basic structure. At a minimum, they must contain a noun (who/what) and a verb (action).
- Spelling is important. There is one (but sometimes multiple) acceptable spelling for a word.

Similarly, Python follows a few different rules.

Rule: New lines are used to separate logical statements (similar to a period at the end of a sentence).

Commentary	**Python code**
Assign x the value 7. Assign y the value 4. Assign z the value of x plus y.	
This is unreadable!	

Rule: Whitespaces (spaces and tabs) are used to indicate which statements should be grouped together and colons (:) are used to indicate the continuation of an idea at a deeper level. Think about this like sub-bullets:

- Dogs are cool
    - They can swim
    - They can bark
    - They can play
- Cats are cool
    - They pur
    - They can jump

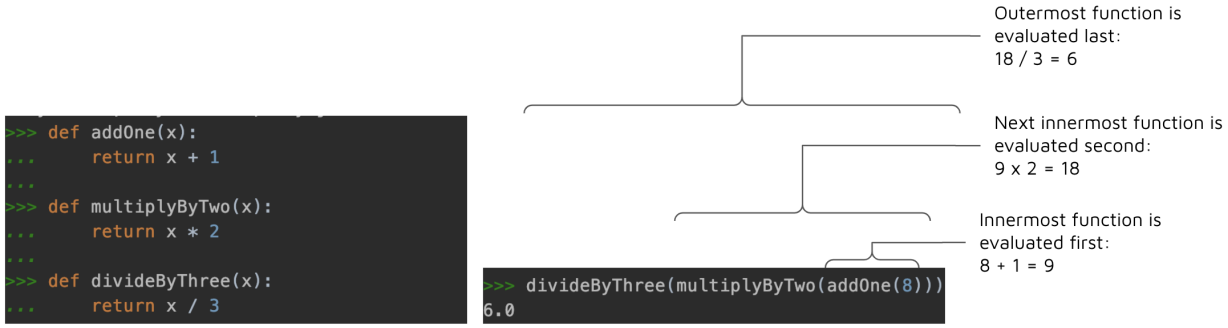
Commentary

- Define a function "someCoolFunction"
  - If the variable "x" is equal to the value 1,
    - print "Something cool"
  - Otherwise
    - print "Something even cooler"

Python code

```
def someCoolFunction():
    if x == 1:
        print('Something cool')
    else:
        print('Something even cooler')
```

Rule: Statements are evaluated from the inside out.



Additional rules will be covered in the variables section.

You can accomplish anything with assignments, conditions, and repetition

- Assignment is linking a value to a name (variable)
- Conditions indicate when to do, or not do, something
- Repetitions do the same thing multiple times

These three basic concepts underpin any coding language and are commonly used in day to day actions (e.g. if it rains today, I will bring my umbrella every time I go outside). Knowing these concepts, means that you already know how to code. Learning Python is just a matter of using the right rules and structures to express what you already know.

Python 2 vs 3

Languages evolve over time to account for new (and hopefully better) ways of expressing information. Python is the same. Python 3 is the newest version of Python and contains a number of breaking changes from Python 2. A breaking change is one that's not backwards compatible. A simple example is the "print" statement. In Python 2, the syntax for a print statement is "print --text--". In Python 3, the syntax is "print(--text--)".

```
>>> print 'this will not work'
  File "<input>", line 1
    print 'this will not work'
                             ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print('this will not work')?
```

In 2020, support for Python 2 was discontinued. Because of the breaking changes, some applications and companies have not updated to Python 3, however, any new applications are all on Python 3 now. Because of this, it's in your best interest to learn Python 3!

Data types

Data type	Python expression	Description	Example(s)	Is mutable
Integer	int	A whole number value	• 1 • 4 • 103987	No
Float	float	A floating point number. Think of it as a decimal.	• 11.234 • 3.14159	No
String	string	A set of characters surrounded by single quotes or double quotes	• “This is a string” • ‘This is also a string’ • ‘Sdjajdn483 293….~03’	No
None	None	Indicates a lack of any data	• None	No

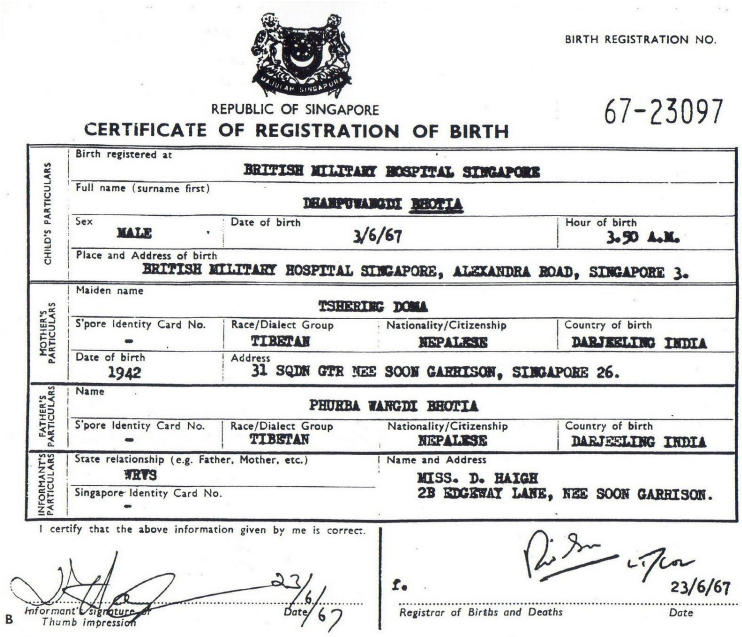
Boolean	bool	A value that evaluates to true or false. Falsy values are: ● Any empty collection, such as: ○ Empty list ○ Empty dictionary ○ Empty string ● The following values: ○ 0 ○ False ○ None Any other values are considered truthy.	● False ● True	No
List	list	An ordered list of values	● [1, 4, 2, 9] ● [1, 'cat', None, 1.4] ● [[1, 2], [3, 4]]	Yes
Tuple	tuple	An immutable, ordered list of values	● (1, 4, 2, 9) ● (1, 'cat', None, 1.4)	No
Dictionary	dict	An unordered list of key, value pairs	● {'apples': 3, 'oranges': 8, 'bananas': 1}	Yes

Variables

A variable is a name for an object. A variable can refer to anything! A variable has a name and a value, expressed as "--name-- = --value--". Variables allow us to:

- Agree on a common term to refer to something
- Create a shorthand way of referring to a complex object

Your name is a variable! A birth certificate assigns you a (somewhat) unique name used to identify you.



BIRTH REGISTRATION NO.

67-23097

REPUBLIC OF SINGAPORE

CERTIFICATE OF REGISTRATION OF BIRTH

CHILD'S PARTICULARS

Birth registered at: BRITISH MILITARY HOSPITAL SINGAPORE

Full name (surname first): DHARPUWANGDI BHOTIA

Sex: MALE | Date of birth: 3/6/67 | Hour of birth: 3.50 A.M.

Place and Address of birth: BRITISH MILITARY HOSPITAL SINGAPORE, ALEXANDRA ROAD, SINGAPORE 3.

MOTHER'S PARTICULARS

Maiden name: TSHERING DOMA

S'pore Identity Card No.: - | Race/Dialect Group: TIBETAN | Nationality/Citizenship: NEPALESE | Country of birth: DARJEELING INDIA

Date of birth: 1942 | Address: 31 SQDN GTR NEE SOON GARRISON, SINGAPORE 26.

FATHER'S PARTICULARS

Name: PHURBA WANGDI BHOTIA

S'pore Identity Card No.: - | Race/Dialect Group: TIBETAN | Nationality/Citizenship: NEPALESE | Country of birth: DARJEELING INDIA

INFORMANT'S PARTICULARS

State relationship (e.g. Father, Mother, etc.): WRVS

Singapore Identity Card No.: -

Name and Address: MISS. D. HAIGH, 2B EDGEWAY LANE, NEE SOON GARRISON.

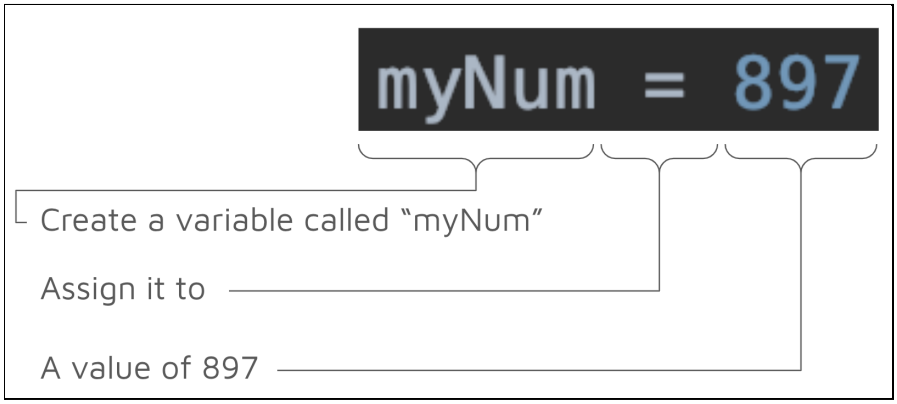
I certify that the above information given by me is correct.

B Informant's Signature or Thumb Impression — Date: 23/6/67

f. Registrar of Births and Deaths — Date: 23/6/67

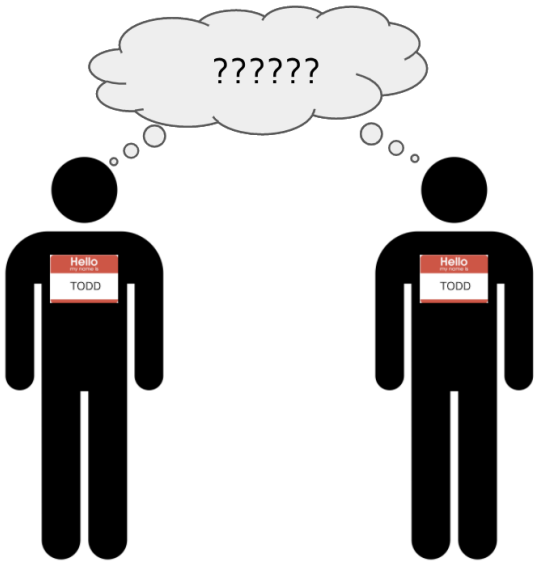
Creating and assigning a variable

The "=" sign is used to assign a variable to a value. It's a common mistake and point of confusion to use the "=" to check for equality, rather than assign a variable. See the section on boolean operations to understand how to check for equality.



Variables must be unique

The same variable cannot refer to two different values at one time. However, a variable can be reassigned to a new value.



Commentary	Python console

"myName" is created and assigned to the value "Todd"	`>>> myName = 'Todd'`
	`>>> myName`
	`'Todd'`

"myName" already exists. The old "myName" variable is destroyed. A new "myName" variable is created and assigned a value of "Imposter".	`>>> myName = 'Imposter'`
	`>>> myName`
	`'Imposter'`

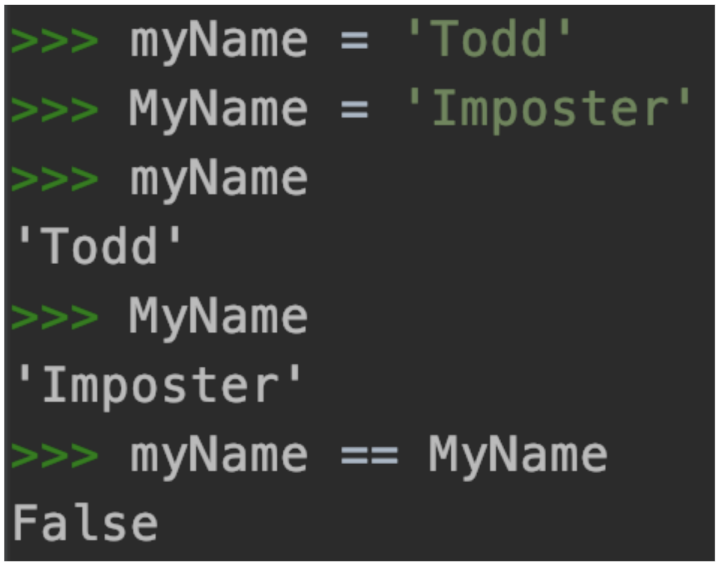
Variable naming hard rules

Rule	Example
Only letters, numbers, and the underscore character (_) can be used	this.is.not.allowed

No spaces	this is not allowed
No quotes	"NotAllowed"
Can't start with a number	6notAllowed

Python is a case sensitive language!

Capitalization matters. Two variables with the same spelling, but different capitalization, are not the same!

Commentary	Python console
"myName" is created and assigned to the value "Todd" "MyName" with a capital "M" doesn't exist yet so it is created and assigned a value of "Imposter" Using an equality check "==" we can see that "myName" and "MyName" are not the same	

Variable naming conventions

There are three types of naming conventions typically used in Python:

Type	Description	Example
Camel case	Starts with a lowercase letter and uses uppercase letters to distinguish words in the variable name	thisIsCamelCase
Pascal case	Starts with a uppercase letter and uses uppercase letters to distinguish words in the variable name	ThisIsPascalCase
Snake case	All lowercase letters with underscores used to distinguish words in the variable name	this_is_snake_case

Pascal case is typically reserved for class names. Nothing will stop you from using Pascal case for variables other than classes, but one of the main points of using variables is to avoid confusion. Others will be confused if you refer to something other than a class using Pascal case.

Using Camel case or Snake case is a matter of personal preference. The important part is to pick one and stick to it. You will notice that I use Camel case throughout my lessons.

Finally always give your variable names meaning. Ask yourself the question - "if some computer illiterate person read my code, would they be able to get a general sense for what it is doing?".

x = 'September 22'

momBirthday = 'September 22'

Variables can be created and changed to any data type

Python is a "dynamic typed" language. This means that you don't have to specify the type of data a variable will contain. Additionally, if you reassign a variable to a different value, the new value doesn't have to be the same data type as the previous one.

```
>>> iAmANumber = 23
>>> iAmANumber = 'still the same variable'
>>> iAmANumber = None
```

[Advanced] Multi-variable assignment

Multiple variables can be created and assigned at the same time with the expression --variable1--, --variable2-- = --iterable with the same length as number of variables--.

```
>>> x, y, z = [1,2,3]
>>> x
1
>>> y
2
>>> z
3
```

Commentary

There are only two variables and 3 values so this causes an error

Python console

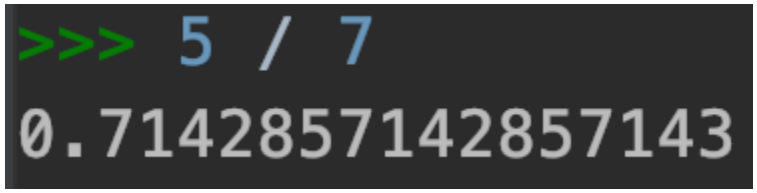
```
>>> x, y = [1,2,3]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

Tuples are another type of iterable that can be used for multi-variable assignment.

```
>>> a, b, c = ('a', 'b', 'c')
```

Number operations

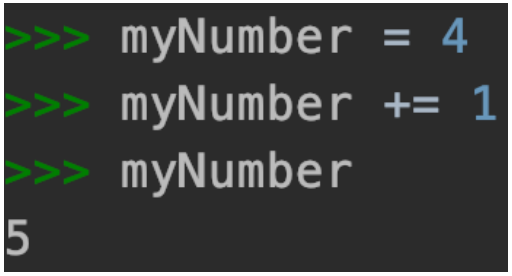
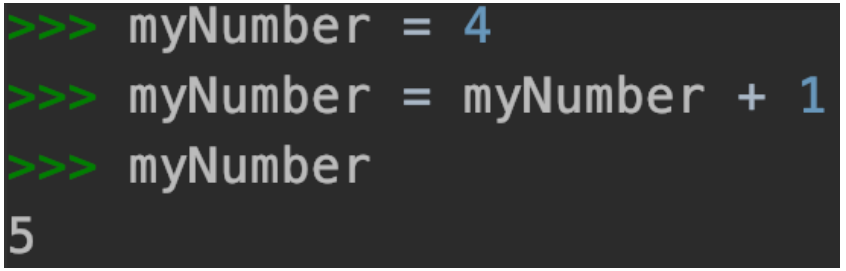
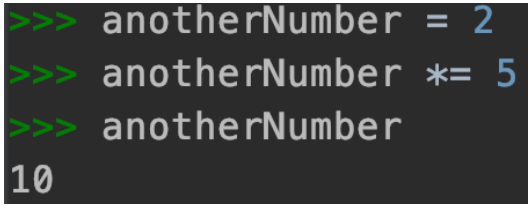
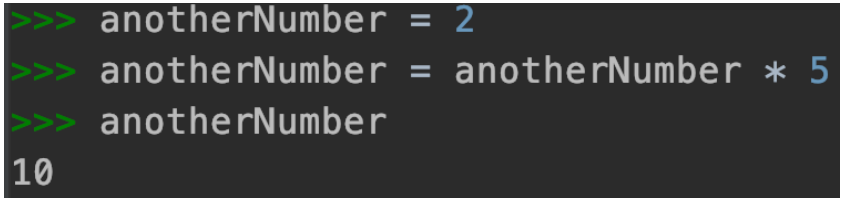
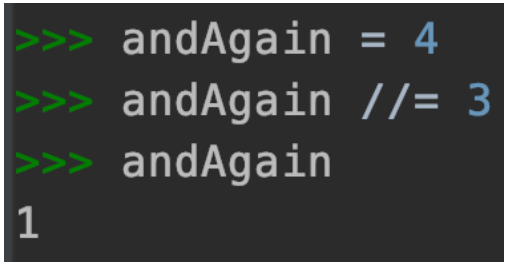
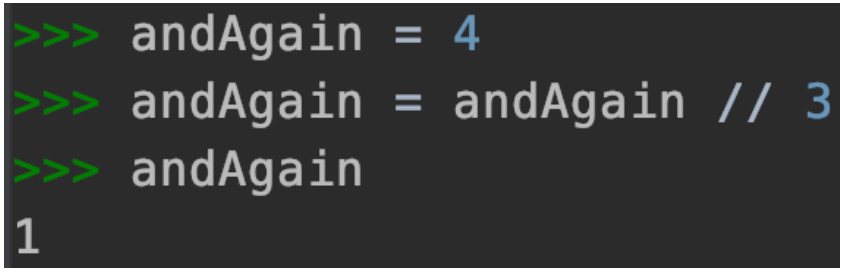
Operation	Description	Example(s)
+	Addition	
-	Subtraction	
*	Multiplication	
**	Exponentiation - raise to the power of	

Operation	Description	Example(s)
/	Division ** Produces a floating point number in Python 3	 
//	Floor division - result of division is rounded down (floored) to the nearest whole number	 
%	Modulo operation - take the remainder of a divided number	

Tips

When performing a number operation on a single variable and re-assigning to the same variable, you can use the shorthand --variable-- --operation--=--number--

Examples

Shorthand	**Regular**
	
	
	

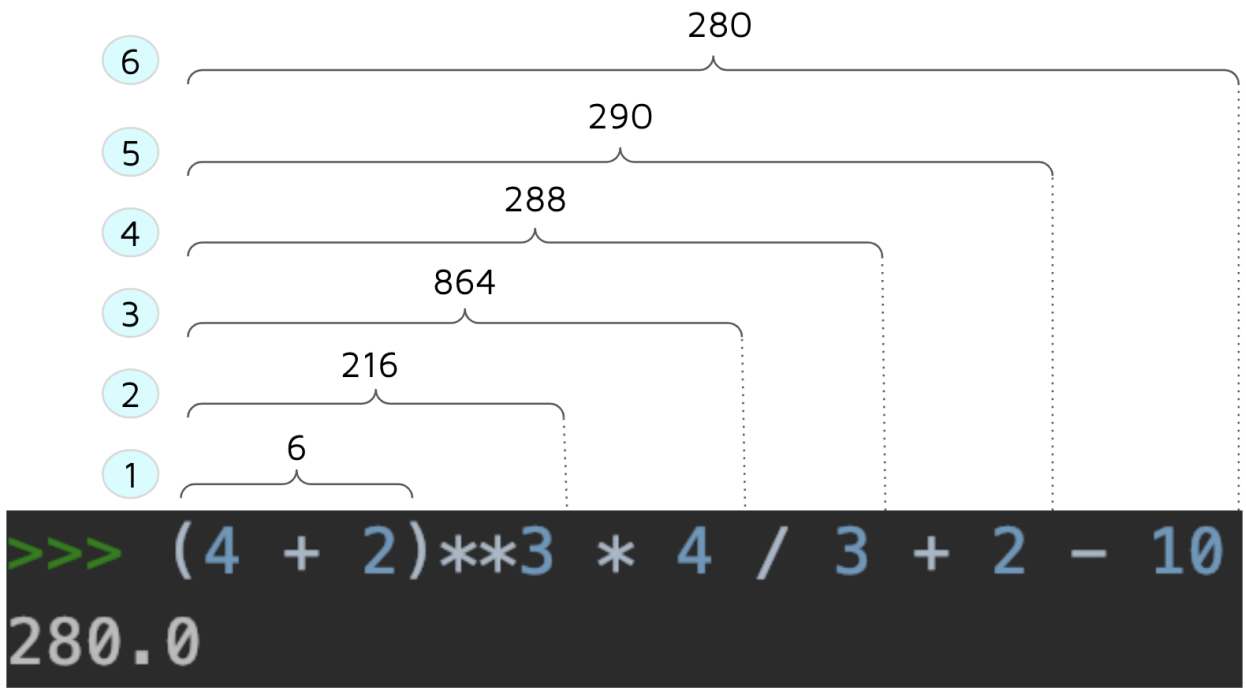
Order of operations

Number operations follow the same order of operations as any mathematical function (PEMDAS):

1. Parenthesis
2. Exponents
3. Multiplication and Division (order between these can differ)
4. Addition and Subtraction (order between these can differ)

When in doubt, add parenthesis! If you're unsure, others probably will be too.

Examples



Booleans

Booleans are a data type that indicates true or false.

Capitalization

Remember that capitalization matters! “True” and “False” are valid boolean values. “true” and “false” are not valid!

Truthy vs Falsey

Truthy and falsey values are any values that are not necessarily a boolean type (True or False), but are considered to be a boolean when asked as a question: “Is there something here?”

Falsey values:

- Any empty collection, such as:
  - Empty list
  - Empty dictionary
  - Empty string
- The following values:
  - 0
  - False
  - None

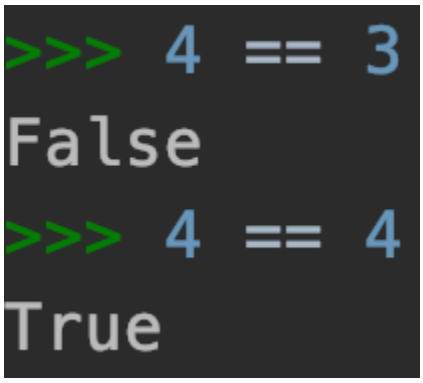
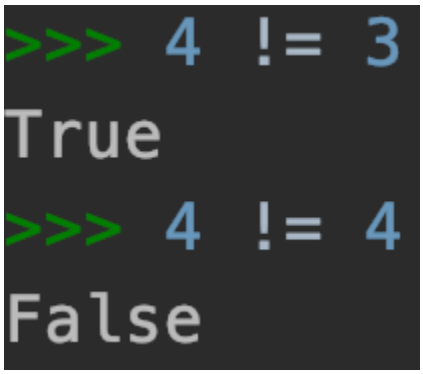
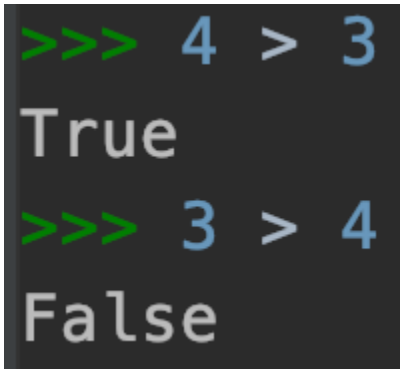
Truthy values:

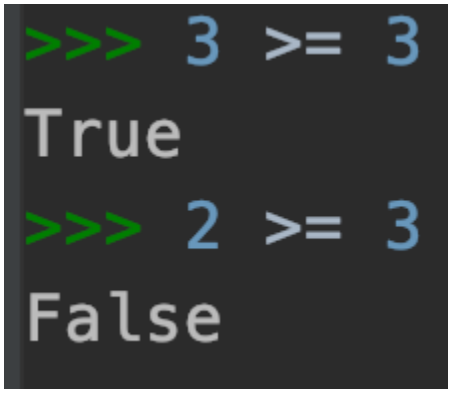
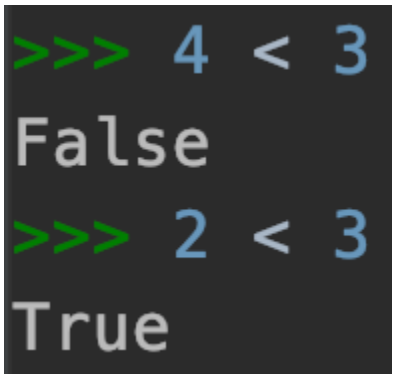
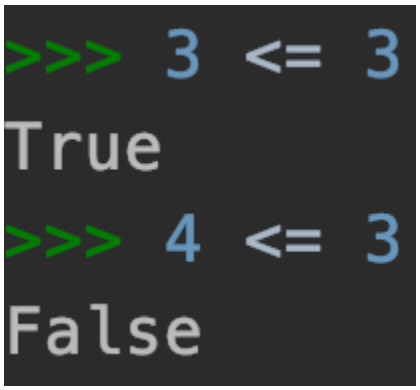
- Anything not considered falsey

Commentary	**Python console**
The “myList” variable is a “list” type, NOT a “boolean” type The “myList” variable evaluated to “false” shown by the fact that the print statement did not execute If we negate the “myList” variable, the print statement does execute. This is synonymous to “if not false, do the following”	

Boolean operations

Think of boolean operations as questions that can be answered by true or false.

Operation	Description	Example(s)
==	Is equal to?	
!=	Is not equal to?	
>	Is greater than?	

Operation	Description	Example(s)
>=	Is greater than or equal to?	
<	Is less than?	
<=	Is less than or equal to?	

Combining booleans

To combine multiple booleans, use "and" and/or "or" operators.

Examples

```
>>> True and False
False
>>> False and False
False
>>> True and True
True
```

```
>>> True or False
True
>>> False or False
False
>>> True or True
True
```

Use parentheses to group booleans together and ensure the order of operations.

Examples

```
>>> (True or False) and (False and True)
False
>>> (True or False or False) and True
True
```

Python is lazy when evaluating multiple booleans, evaluating from left to right.

Examples

Commentary	Python console
"True or {anything}" will always evaluate to "True". Python knows this and never executes the rest of the statement	`>>> True or error` `True`

"True and {anything}" requires the second part of the statement to be executed.	`>>> True and error`
	`Traceback (most recent call last):`
	`  File "<input>", line 1, in <module>`
	`NameError: name 'error' is not defined`

Negating booleans

To negate (take the opposite) of a boolean, use the "not" operator.

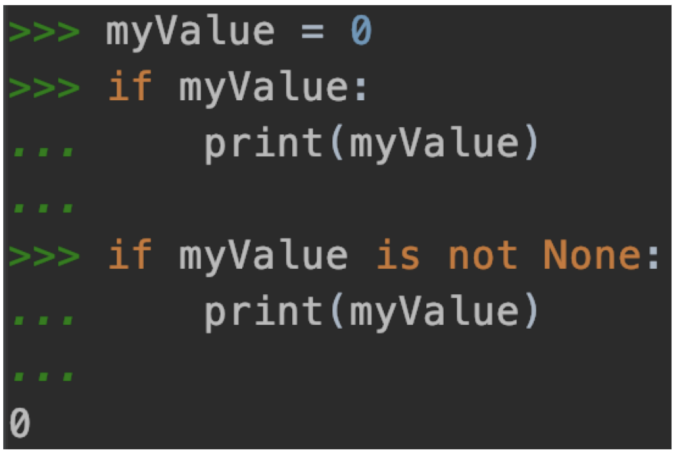
Examples

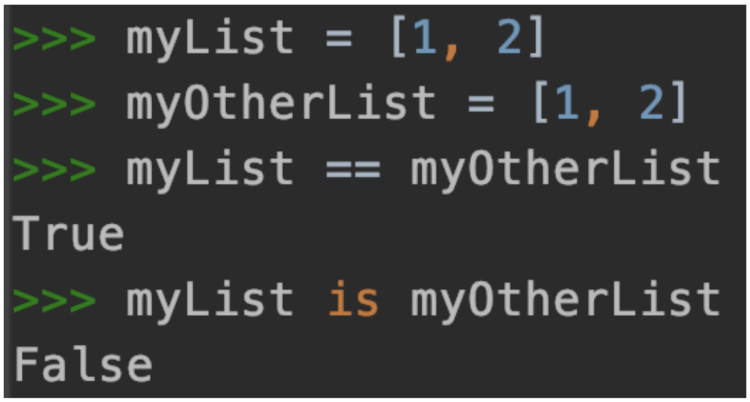
```
>>> not False
True
>>> not True
False
```

Using "Is"

Most of the time, we care about comparing values against each other. The "is" operator compares the underlying object. The "is" operator is useful when you care about the absence of data or you want to compare mutable objects like lists or dictionaries.

Examples

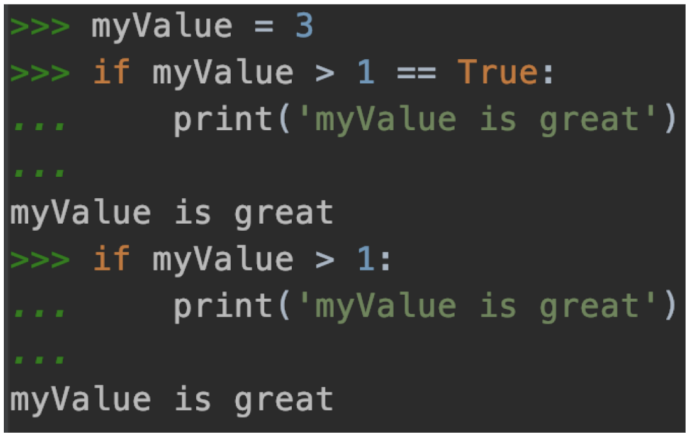
Commentary	Python console
A value of 0 is falsey Because 0 is falsey, the print statement will not execute If we only case about a value of None, we can compare 0 with None. These are not the same object so we print the value of 0	

Commentary	Python console
The values of these two lists are equal These two lists are two different “boxes” stored in different areas of your computer’s memory	

Boolean tips

There is no need to compare a truthy or falsey value with True or False.

Examples

Commentary	Python console
This is equivalent to saying "if True equals True"	
This is a better, shorter way of expressing the same logic	

```
>>> myValue = 3
>>> if myValue > 1 == True:
...     print('myValue is great')
...
myValue is great
>>> if myValue > 1:
...     print('myValue is great')
...
myValue is great
```

Conditionals

A conditional expresses an action that will happen only if (on condition) something is true. In common language, we express conditionals all the time:

- If the water is warm, I will go swimming
- If "I Love You Man" is on TV and my girlfriend wants to watch, I will make popcorn

Because conditionals check whether something is true, they necessarily rely on boolean values (true/false).

If statement

Expressed as:

```
>>> if {condition}:
...     {do this}
```

Any number of conditions can be included.

```
>>> x = 3
>>> y = 4
>>> if x > y or x == y or False or y > 2:
...     print('That was a long if statement')
...
That was a long if statement
```

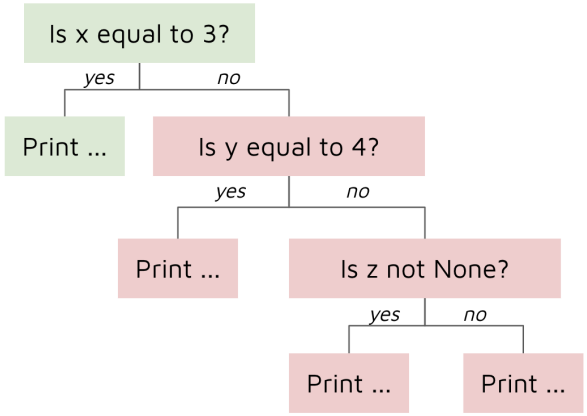
If statements can be "nested" as many times as needed.

```
>>> x = 3
>>> y = 4
>>> z = []
>>> if z is not None:
...     if y > x:
...         if x == 3:
...             print('Nested like Russian dolls')
...
Nested like Russian dolls
```

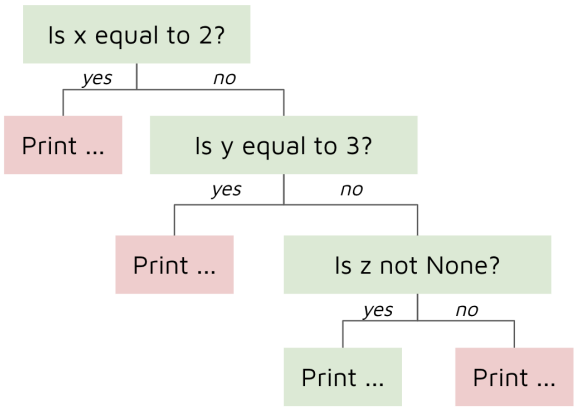
If, elif, else statement

"elif" and "else" are optional additions to an "if" statement, allowing greater control over conditional flow. Think of it as a decision tree. The first "if" or "elif" statement which evaluates to true will execute. No other statements will execute. If none of the "if" or "elif" statements are true, the "else" statement will execute. There can only be one "if" and one "else", but there can be any number of "elif"s, or none at all.

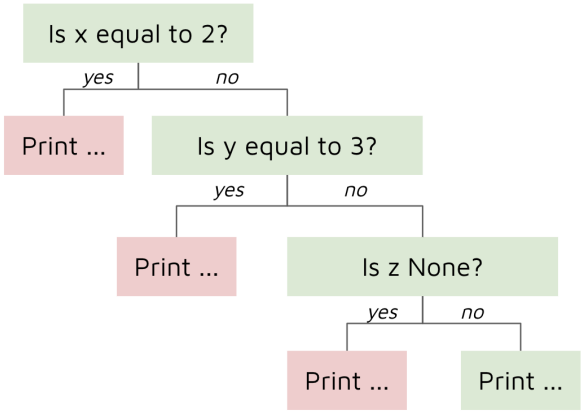
```
>>> x = 3
>>> y = 4
>>> z = []
>>> if x == 3:
...     print('The first statement executed')
... elif y == 4:
...     print('The second line executed')
... elif z is not None:
...     print('The third line executed')
... else:
...     print('The fourth line executed')
...
The first statement executed
```



```
>>> if x == 2:
...     print('The first statement executed')
... elif y == 3:
...     print('The second line executed')
... elif z is not None:
...     print('The third line executed')
... else:
...     print('The fourth line executed')
...
The third line executed
```

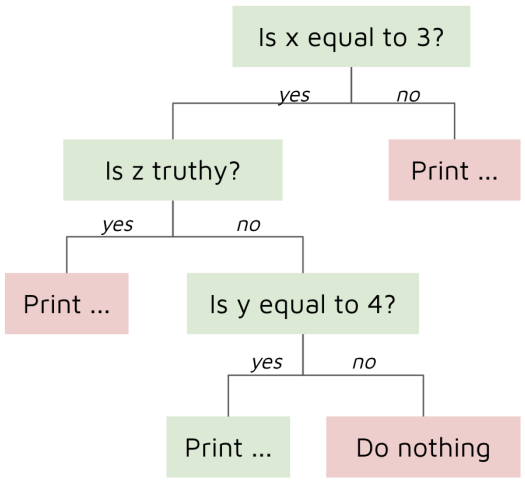



```
>>> if x == 2:
...     print('The first statement executed')
... elif y == 3:
...     print('The second line executed')
... elif z is None:
...     print('The third line executed')
... else:
...     print('The fourth line executed')
...
The fourth line executed
```

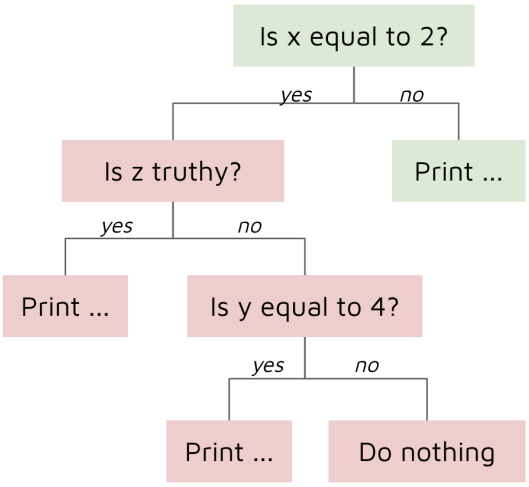


If, elif, else statements can be nested just like if statements.

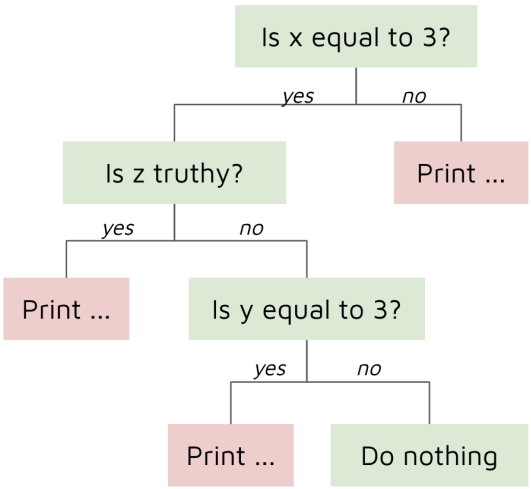
```
>>> x = 3
>>> y = 4
>>> z = 0
>>> if x == 3:
...     if z:
...         print('The first statement executed')
...     elif y == 4:
...         print('The second statement executed')
... else:
...     print('The third line executed')
...
The second statement executed
```



```
>>> if x == 2:
...     if z:
...         print('The first statement executed')
...     elif y == 4:
...         print('The second statement executed')
... else:
...     print('The third line executed')
...
The third line executed
```

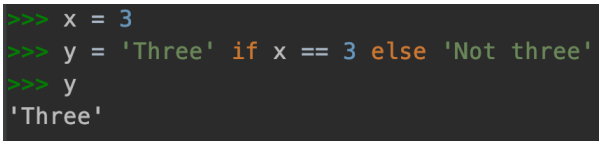
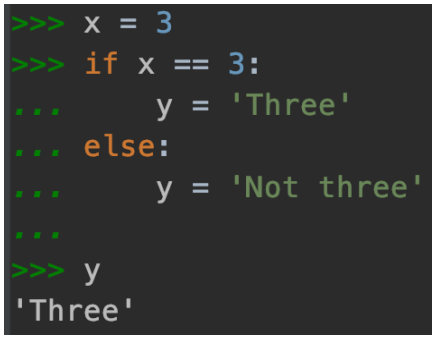
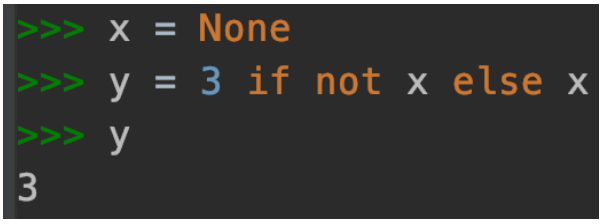
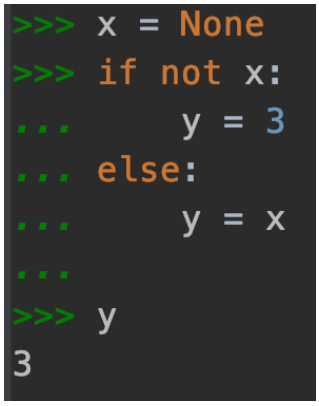



```
>>> if x == 3:
...     if z:
...         print('The first statement executed')
...     elif y == 3:
...         print('The second statement executed')
... else:
...     print('The third line executed')
...
```



[Advanced] Ternary operators

Ternary operators are a shorthand way of writing "if" statements. The benefit can be more readable code in fewer lines. Like "if" statements, ternary operators can be nested, but readability declines as more nesting is used.

Ternary operator	Normal "if" statement
	
	

Lists and dictionaries

Lists and dictionaries have some common properties:

- They are examples of "iterables", meaning they contain 0 to infinite items that can be selected one by one (e.g. they can be iterated over)
- They are mutable, meaning they can be modified after creation
- They can contain anything, even other lists or dictionaries
- They both act as a way to create logical groupings of data and refer to it in a single variable

They are also different in some ways:

Lists	Dictionaries
Created with brackets: []	Created with braces: {}
Contain an ordered list of values	Contain an unordered list of values
Contain just values	Contain key, value pairs

Values are accessed by using the reference to the position in the list (the index)	Values are accessed by using the key which is mapped to the value

Think of lists or dictionaries as "boxes" to put items in. When we assign a list or dictionary to a variable, we are assigning the "box", not the contents of the box. When we add, update, or delete an item from a list or dictionary, we have to provide the location within the box; either the position of a list or the key of a dictionary.

Examples of logical groupings using lists or dictionaries:

Commentary	**Python code**

Items 1-3 are all logically connected, but expressed in individual variables. Imagine giving your partner a separate piece of paper for every item to buy from the grocery store
 	
Instead, we can include all items in on single list
	`# Grocery store items`
	`item1 = 'apples'`
	`item2 = 'milk'`
	`item3 = 'shampoo'`
	
	`groceryStoreItems = ['apples', 'milk', 'shampoo']`

It would be even worse if you gave a separate piece of paper for each item and another paper for the quantity to buy	`# Grocery store items`
	`item1 = 'apples'`
	`item1Quantity = 2`
	`item2 = 'milk'`
	`item2Quantity = 1`
	`item3 = 'shampoo'`
	`item3Quantity = 1`
	
	`groceryStoreItems = {'apples': 2, 'milk': 1, 'shampoo': 1}`

Dictionaries can save your relationship	

Mutation

Lists and dictionaries can mutate, meaning they can be modified after creation. Using the "box" analogy, we can add, change, or remove items from the box, but the box itself will remain the same.

Commentary	Python console

Lists are mutable, tuples are not
 	
Updating the first item in the list is perfectly fine
 	
Updating the first item in the tuple causes an error
	`>>> myMutableList = [1, 2, 3]`
	`>>> myStaticTuple = (1, 2, 3)`
	`>>> myMutableList[0] = 3`
	`>>> myStaticTuple[0] = 3`
	`Traceback (most recent call last):`
	`  File "<input>", line 1, in <module>`
	`TypeError: 'tuple' object does not support item assignment`

Lists overview

Lists, commonly referred to as "arrays" in other programming languages, contain any number of ordered items.

Create a list

Lists can be created in two ways: bracket notation or referring to the list class. I almost always use the bracket notation.

```
thisIsAList = []
thisIsAlsoAList = list()
```

You can also add items to your list when you create it. Items in a list are separated by commas.

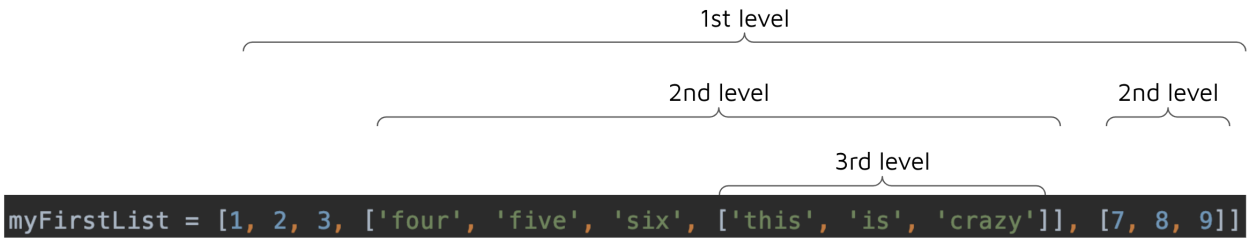
```
thisListHasItems = [1, 2, 3]
```

Lists do not have to contain homogeneous items

```
myVariable = 2.71
thisIsAWildList = [1, 'someThing', myVariable]
```

Lists can be nested

Nesting is when you put a container inside of another container. A common analogy is Russian nesting dolls. Lists can be nested as many times as you want, but accessing the values that are in the deeper levels of nesting will become more complicated.



Lists begin at index 0!!!

The first item in a list is at index 0. This can take some time to get used to and is a common cause for mistakes for those new to programming.

Access items in a list

Item(s) are accessed by using bracket notation.

A single item can be accessed using [--position in list--].

```
>>> myList = ['first item', 'second item', 'third item']
>>> myList[0]
'first item'
>>> myList[1]
'second item'
```

Negative numbers can be used to access items starting at the end of the list:

```
>>> myList = ['first item', 'second item', 'third item']
>>> myList[-1]
'third item'
>>> myList[-2]
'second item'
```

An error will occur if you provide an index greater than the index of the last item.

```
>>> myList = ['first item', 'second item', 'third item']
>>> myList[5]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list index out of range
```

Multiple items can be accessed using [--starting position in list--, --ending position in list--]. The starting position is **inclusive**, but the ending position is **exclusive**. When multiple items are accessed, the value provided is another list containing the selected items, as opposed to just the value of the item when a single item is accessed.

```
>>> myList = ['first item', 'second item', 'third item']
>>> myList[0:3] # Read as "from index 0 to the index before 3 (2)"
['first item', 'second item', 'third item']
>>> myList[0:2] # Read as "from index 0 to the index before 2 (1)"
['first item', 'second item']
```

Either or both the starting and ending position can be left blank. If the starting position is left blank, index 0 (the start of the list) is assumed. If the ending position is left blank, the length of the list is assumed (which would select the final item in the list).

```
>>> myList = ['first item', 'second item', 'third item']
>>> myList[:2]
['first item', 'second item']
>>> myList[1:]
['second item', 'third item']
>>> myList[:]
['first item', 'second item', 'third item']
```

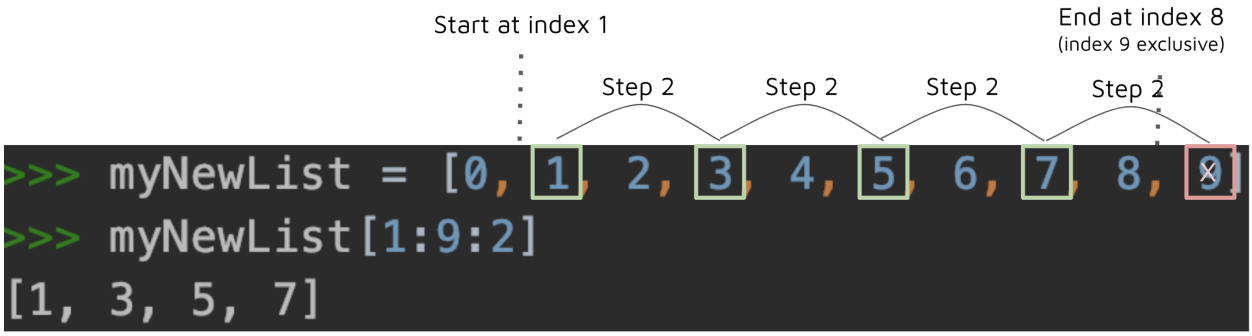
Negative numbers can be used here as well. Think of the negative number as acting on the length of the list. In the example below, the length of the list is 3 so you can think of this as [0:(3-1)] or [0:2]. Both are equivalent.

```
>>> myList = ['first item', 'second item', 'third item']
>>> myList[0:-1]
['first item', 'second item']
```

The same logic applies for the starting position. In the example below, the length of the list is 3 so you can think of this as [(3-1):3] or [2:3]

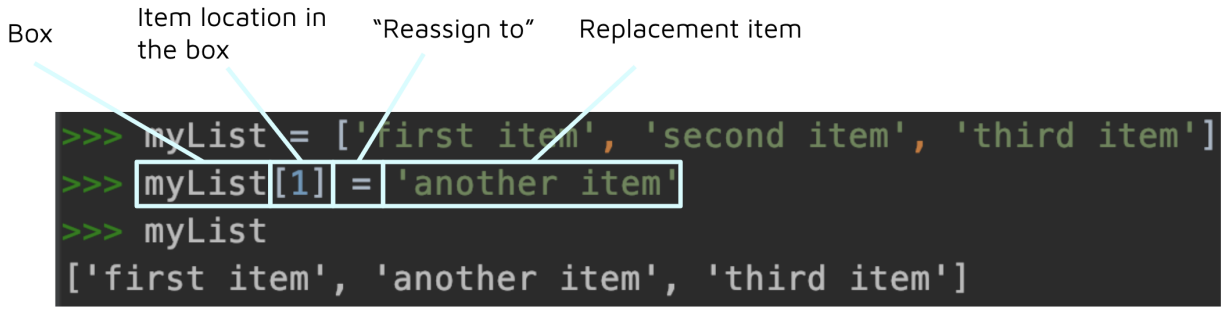
```
>>> myList = ['first item', 'second item', 'third item']
>>> myList[-1:3]
['third item']
```

You can also add a third number to the list accessor to set the "step" size. The step is the number of indexes to move before selecting another item.



Update items in a list

Items can be updated by referencing their index in the list and assigning to a new value. To continue with the box analogy, first we identify the "box" we want to update, then we identify the location of the item in the box we want to replace, then we specify the item that should replace it.



You can only update items that exist.

```
>>> myList = ['first item', 'second item', 'third item']
>>> myList[4] = 'outside of box'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list assignment index out of range
```

To add an item, see the section on list methods and functions.

Common list methods and functions

There are plenty of list functions and methods that are helpful in certain situations. This will cover some of the most common. It is helpful, but not necessary, to understand the difference between functions and methods before reviewing this material.

Methods and functions that add item(s) to a list

- **append** - (method) add one item to the end of a list
- **Insert** - (method) add on item to a specified index in a list
- **extend** - (method) add a list onto the existing one
- **+** - (function) add two lists together

```
>>> myList = ['first item', 'second item', 'third item']
>>> myList.append('fourth item')
>>> myList
['first item', 'second item', 'third item', 'fourth item']
```

```
>>> myList = ['first item', 'second item', 'third item']
>>> myList.insert(1, 'fourth item')
>>> myList
['first item', 'fourth item', 'second item', 'third item']
```

```
>>> myList = ['first item', 'second item', 'third item']
>>> myList.extend(['fourth item'])
>>> myList
['first item', 'second item', 'third item', 'fourth item']
>>> myList.extend([1, 2])
>>> myList
['first item', 'second item', 'third item', 'fourth item', 1, 2]
```

```
>>> myList = ['first item', 'second item', 'third item']
>>> myList + [1, 2] # Returns the value, but doesn't modify myList
['first item', 'second item', 'third item', 1, 2]
>>> myList
['first item', 'second item', 'third item']
>>> myList += [1, 2] # Assigns the returned value to myList
>>> myList
['first item', 'second item', 'third item', 1, 2]
```

Methods and functions that remove item(s) from a list

- **pop** - (method) remove a value from a list and return it
- **remove** - (method) remove a value from a list. Does not return anything
- **del** - (function) remove item(s) from a list using index(es)

```
>>> myList = ['first item', 'second item', 'third item']
>>> myItem = myList.pop(1)
>>> myItem
'second item'
>>> myList
['first item', 'third item']
```

```
>>> myList = ['first item', 'second item', 'third item']
>>> myList.remove('third item')
>>> myList
['first item', 'second item']
```

```
>>> myList = ['first item', 'second item', 'third item']
>>> del myList[0]
>>> myList
['second item', 'third item']
>>> myList = ['first item', 'second item', 'third item']
>>> del myList[1:3]
>>> myList
['first item']
```

Methods and functions that sort a list

- **sort** - (method) sort the elements in a list
- **sorted** - (function) copy the existing list, sort the elements, and return the new list

```
>>> myAlphabeticList = ['oranges', 'pears', 'apples', 'kumquat']
>>> myNumericList = [2, 31, 16.5, 3.14, 10001]
>>> myAlphabeticList.sort()
>>> myAlphabeticList
['apples', 'kumquat', 'oranges', 'pears']
>>> myNumericList.sort()
>>> myNumericList
[2, 3.14, 16.5, 31, 10001]
```

```
>>> myAlphabeticList = ['oranges', 'pears', 'apples', 'kumquat']
>>> sorted(myAlphabeticList)
['apples', 'kumquat', 'oranges', 'pears']
>>> myAlphabeticList # Not in sort order because only a new list was sorted
['oranges', 'pears', 'apples', 'kumquat']
>>> mySortedAlphabeticList = sorted(myAlphabeticList)
>>> mySortedAlphabeticList # This list is sorted because we assigned it the returned value of sorted()
['apples', 'kumquat', 'oranges', 'pears']
```

- **len** - (function) returns the count of items in the list

```
>>> myList = ['first item', 'second item', 'third item']
>>> len(myList)
3
```

Check if an item is in a list

You can check whether an item is in a list using the syntax "--item-- in --list--". This is useful when used with a conditional statement.

```
>>> myList = ['first item', 'second item', 'third item']
>>> 'first item' in myList
True
>>> if 'fourth item' in myList:
...     print('Found item')
... else:
...     print('Not in list')
...
Not in list
>>> if 'third item' in myList:
...     print('Found item')
... else:
...     print('Not in list')
...
Found item
```

[Advanced] List comprehension

List comprehension allows you to do the following in a single line:

- Create a new list
- Loop through an existing list
- Use conditional(s)
- Append to the new list

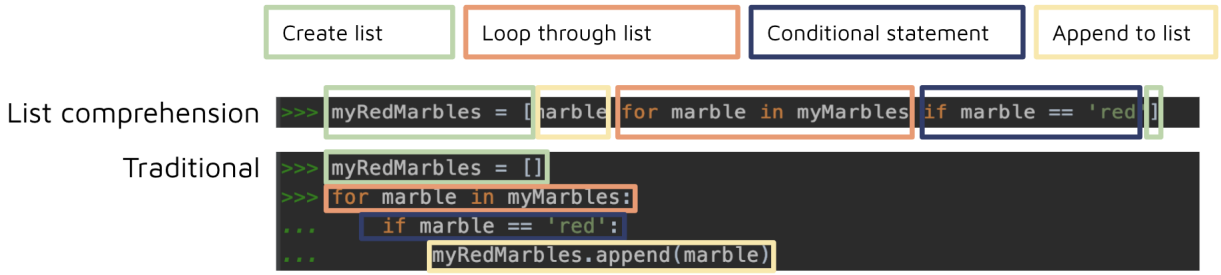
This provides a more concise way of filtering a list. List comprehension is expressed as "[--item-- for --item-- in --iterable-- if --condition(s)--]".

Commentary	Python console

Single line list comprehension	`>>> myMarbles = ['red', 'green', 'red', 'red', 'blue', 'purple', 'green']`
	`>>> myRedMarbles = [marble for marble in myMarbles if marble == 'red']`
	`>>> myRedMarbles`
	`['red', 'red', 'red']`

A traditional loop has the same result, but takes 4 lines of code	`>>> myRedMarbles = []`
	`>>> for marble in myMarbles:`
	`...     if marble == 'red':`
	`...         myRedMarbles.append(marble)`
	`...`
	`>>> myRedMarbles`
	`['red', 'red', 'red']`

When you break down each part of the list comprehension, it aligns with a unique part of a traditional loop.



The conditional part of the list comprehension is optional.

```
>>> numbers = [1, 2, 3, 4]
>>> biggerNumbers = [num + 5 for num in numbers]
>>> biggerNumbers
[6, 7, 8, 9]
```

Just like with traditional loops, you can nest list comprehensions. However, nested list comprehensions become much less readable and understandable. It's a judgment call, but if you need to use a nested list comprehension, you should consider using a traditional loop.

```
>>> numbers = [1, 2, 3, 4]
>>> multiplier = [2, 4]
>>> multipliedNumbers = [x * y for x in numbers for y in multiplier]
>>> multipliedNumbers
[2, 4, 4, 8, 6, 12, 8, 16]
>>> multipliedNumbers = []
>>> for x in numbers:
...     for y in multiplier:
...         multipliedNumbers.append(x * y)
...
>>> multipliedNumbers
[2, 4, 4, 8, 6, 12, 8, 16]
```

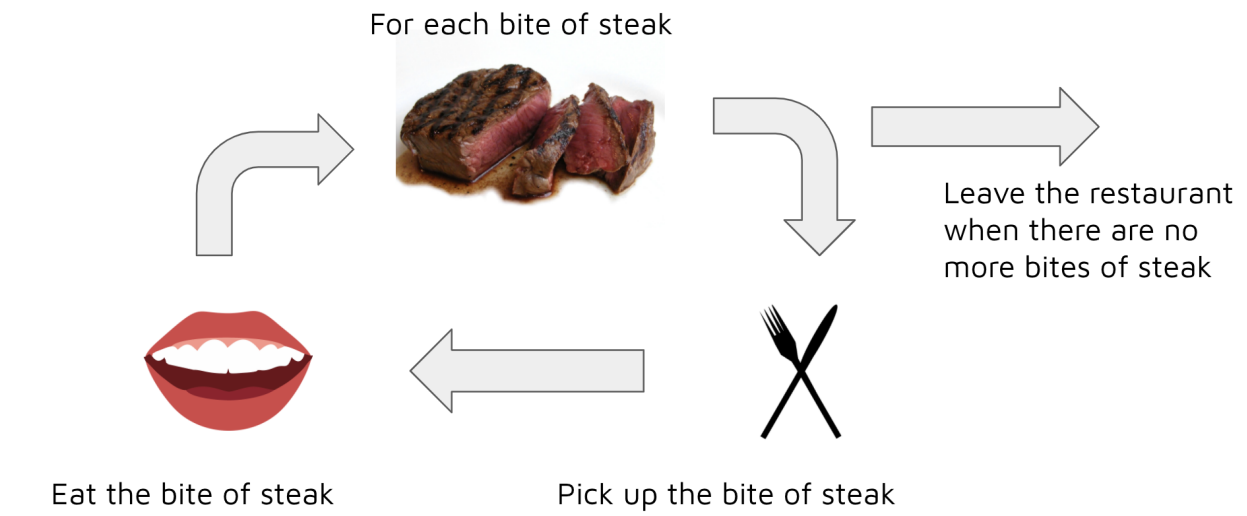
Loops / Iteration

Lists and dictionaries are examples of iterables, meaning you can select items one by one. Loops allow you to iterate through iterables and do something with each item. Remarkably, everything in your life can be expressed as a loop. Loops simply express the conditions when to start and stop an activity. Python provides two types of loops:

- "For" loops are always used with iterables like lists and dictionaries. They allow you to select each item one by one and do something with the item. The loop terminates once you reach the end of the iterable. This is way more common than a "while" loop.
- "While" loops can be used with iterables, but don't need to be. While loops terminate when a condition becomes false.

For loops

Expressed as "for --item-- in --iterable--:"




```
>>> bitesOfSteak = ['bite 1', 'bite 2', 'bite 3', 'bite 4', 'bite 5']
>>> for bite in bitesOfSteak:
...     print(f'Chewing {bite} of steak')
...
Chewing bite 1 of steak
Chewing bite 2 of steak
Chewing bite 3 of steak
Chewing bite 4 of steak
Chewing bite 5 of steak
```

While loops

Expressed as "while --condition(s)--:". While loops can execute forever if the condition(s) never change to False!

```
>>> bitesOfSteak = ['bite 1', 'bite 2', 'bite 3', 'bite 4', 'bite 5']
>>> bitesTaken = 0
>>> while bitesTaken < len(bitesOfSteak):
...     print(f'Chewing {bitesOfSteak[bitesTaken]} of steak')
...     bitesTaken += 1
...
Chewing bite 1 of steak
Chewing bite 2 of steak
Chewing bite 3 of steak
Chewing bite 4 of steak
Chewing bite 5 of steak
```

bitesTaken	len(bitesOfSteak)	Is bitesTaken < len(bitesOfSteak)
0	5	True
1	5	True
2	5	True
3	5	True
4	5	True
5	5	False

Here is an example with multiple conditions. You can provide any number of conditions that can be reduced to True or False.

```
>>> bitesOfSteak = ['bite 1', 'bite 2', 'bite 3', 'bite 4', 'bite 5']
>>> bitesTaken = 0
>>> isHungry = True
>>> while bitesTaken < len(bitesOfSteak) and isHungry:
...     print(f'Chewing {bitesOfSteak[bitesTaken]} of steak')
...     bitesTaken += 1
...     if bitesTaken == 3:
...         isHungry = False
...
Chewing bite 1 of steak
Chewing bite 2 of steak
Chewing bite 3 of steak
```

bitesTaken	len(bitesOfSteak)	isHungry	Is bitesTaken < len(bitesOfSteak)
0	5	True	True
1	5	True	True
2	5	True	True
3	5	False	True
4	5	False	True
5	5	False	False

Nesting

Loops can be nested as many times as you want. You can even combine for and while loops in a nested structure.

```
>>> bitesOfSteak = ['bite 1', 'bite 2', 'bite 3', 'bite 4', 'bite 5']
>>> numberOfChews = [1, 2, 3]
>>> for bite in bitesOfSteak:
...     for chew in numberOfChews:
...         print(f'Chewing {bite} of steak')
...
Chewing bite 1 of steak
Chewing bite 1 of steak
Chewing bite 1 of steak
Chewing bite 2 of steak
Chewing bite 2 of steak
Chewing bite 2 of steak
Chewing bite 3 of steak
Chewing bite 3 of steak
Chewing bite 3 of steak
Chewing bite 4 of steak
Chewing bite 4 of steak
Chewing bite 4 of steak
Chewing bite 5 of steak
Chewing bite 5 of steak
Chewing bite 5 of steak
```

[Advanced] Control flow in loops

The “continue” and “break” statements provide more fine grained control over a loop.

Continue

The “continue” statement allows you to skip the rest of the code in the current loop and continue to the next item in the list.

```
>>> myMarbles = ['red', 'green', 'red', 'red', 'blue', 'purple', 'green']
>>> myRedMarbles = []
>>> myOtherMarbles = []
>>> for marble in myMarbles:
...     if marble == 'red':
...         myRedMarbles.append(marble)
...         continue  # Stop the code here if this is a red marble
...     myOtherMarbles.append(marble)  # This will only run if the marble is NOT red
...
>>> myRedMarbles
['red', 'red', 'red']
>>> myOtherMarbles
['green', 'blue', 'purple', 'green']
```

Break

The break statement will execute the loop entirely.

```
>>> minesweeper = ['safe', 'safe', 'safe', 'mine', 'safe', 'safe']
>>> for tile in minesweeper:
...     if tile == 'safe':
...         print('No mine here')
...     else:
...         print('You hit a mine!')
...         break
...
No mine here
No mine here
No mine here
You hit a mine!
```

[Advanced] Enumerate function

The enumerate function allows you to access the current index of an iterable while using a "for" loop. It's important to understand how multi-variable assignment works to grasp how enumeration works. Enumeration can be helpful for a few reasons. The one that I use it for the most is to access another list using the index of the list I'm looping through (example below).

```
>>> myShoppingList = ['apples', 'oranges', 'bananas']
>>> myQuantityList = [4, 3, 10]
>>> for idx, fruit in enumerate(myShoppingList):
...     quantityToBuy = myQuantityList[idx]
...     print(f'Buy {quantityToBuy} {fruit}')
...
Buy 4 apples
Buy 3 oranges
Buy 10 bananas
```

Note that in the above example, it would be better to express the shopping list as a dictionary with the fruits as keys and quantities as values. This is just used to illustrate enumeration with a simple example.

Under the hood, the enumerate function creates an index, item pair for each item in the list.

```
>>> myShoppingList = ['apples', 'oranges', 'bananas']
>>> list(enumerate(myShoppingList))
[(0, 'apples'), (1, 'oranges'), (2, 'bananas')]
```

By using multi-assignment, the first item (the index) gets assigned to the "idx" variable and the second item (the original item in the list) gets assigned to "fruit".

```
(0, 'apples')
(1, 'oranges')
(2, 'bananas')
```

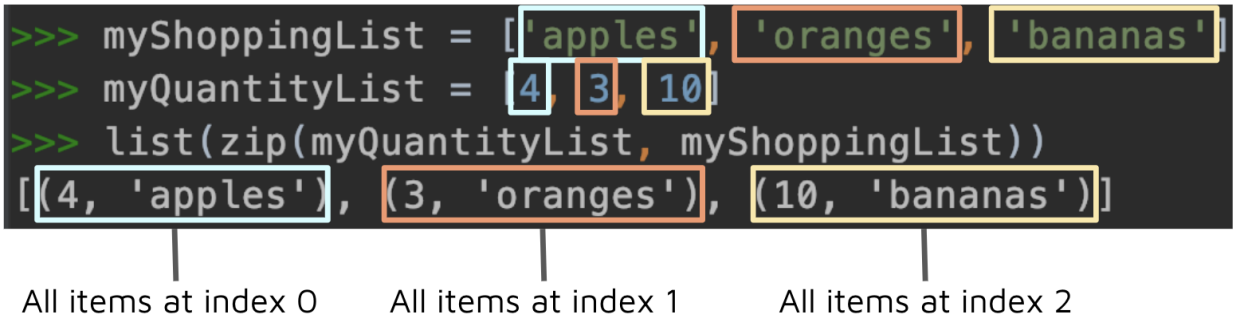
```
>>> for idx, fruit in enumerate(myShoppingList):
...     quantityToBuy = myQuantityList[idx]
...     print(f'Buy {quantityToBuy} {fruit}')
```

[Advanced] Zip function

The zip function is pretty similar to the enumerate function except it allows you to combine two or more items into the same iterable. Just like the enumerate function, it's important to understand multi-variable assignment.

```
>>> myShoppingList = ['apples', 'oranges', 'bananas']
>>> myQuantityList = [4, 3, 10]
>>> for quantity, fruit in zip(myQuantityList, myShoppingList):
...     print(f'Buy {quantity} {fruit}')
...
Buy 4 apples
Buy 3 oranges
Buy 10 bananas
```

Under the hood, the zip function is taking every item at the same index and combining it into a tuple. Just like a coat zipper takes the "teeth" from opposite sides and combines them in the middle.



You can combine as many iterables as you want, but they all have to be the same length.

```
>>> myShoppingList = ['apples', 'oranges', 'bananas']
>>> myQuantityList = [4, 3, 10]
>>> fruitColor = ['red', 'orange', 'yellow']
>>> list(zip(myQuantityList, myShoppingList, fruitColor))
[(4, 'apples', 'red'), (3, 'oranges', 'orange'), (10, 'bananas', 'yellow')]
```

[Advanced] Map function

The map function applies a function to every item within a list.

```
>>> def claimItem(itemToBeClaimed):
...     return f'My {itemToBeClaimed}'
...
>>> myList = ['milk', 'butter', 'juice']
>>> myList = list(map(claimItem, myList))
>>> myList
['My milk', 'My butter', 'My juice']
```

There are a couple things that may seem odd about the example above. First, the "list" function is explicitly called on the result of the "map" function. This is because the map function does not return a list, but rather a different type of iterable known as a generator.

Generators are beyond the scope of this material, but just know that any iterable, including generators, can be "forced" to become a list.

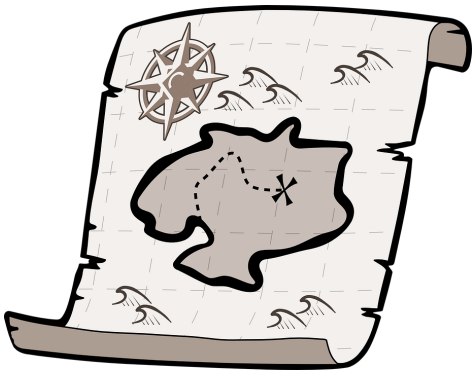
Secondly, the "claimItem" function is used as the first argument in the "map" function without parentheses. See the section on functions as objects to understand why that is the case.

Here is the long way to achieve the same result of a map function:

```
>>> def claimItem(itemToBeClaimed):
...     return f'My {itemToBeClaimed}'
...
>>> myItems = []
>>> for item in ['milk', 'butter', 'juice']:
...     myItems.append(claimItem(item))
...
>>> myItems
['My milk', 'My butter', 'My juice']
```

Dictionaries overview

Dictionaries contain any number of **unordered** sets of key/value pairs. This can be difficult to think about at first, especially when the first thing that comes to mind is a literal, physical dictionary which is ordered alphabetically. It's better to think about Python dictionaries as a treasure map. Each key is an "X" on the map which leads you to the value. The "X"s can be anywhere on the map, but they lead you to the treasure every time.



Create a dictionary

Dictionaries can be created in two ways: braces notation or referring to the dict (dictionary) class. I almost always use the braces notation.

```
>>> thisIsADictionary = {}
>>> thisIsAlsoADictionary = dict()
```

You can also add key/value pairs to a list when you create it.

```
>>> thisIsADictionary = {'CO': 'Colorado', 'MT': 'Montana', 'RI': 'Rhode Island'}
```

Dictionaries do not have to contain homogeneous values

```
>>> aVariable = 27
>>> myDict = {'a': True, 'b': None, 'squirrel': aVariable, 'banana': 0.0001}
```

Dictionary keys can be any immutable object

Typically, dictionary keys are strings, but they don't have to be!

Commentary	**Python console**
Any immutable object can be a dictionary key	`>>> myDict = {3: 'three', 3.7: 'three point seven', True: 8, None: 0}`

A list is mutable so we get an error when we try to use it as a key	`>>> myDict = {[1, 22]: 'this won\'t work'}`
	`Traceback (most recent call last):`
	`  File "<input>", line 1, in <module>`
	`TypeError: unhashable type: 'list'`

Each key in the dictionary must be unique

Commentary	**Python console**

Keys must be unique, if the same value is used twice, only one will be saved	`>>> confusingDict = {'I am Todd': 27, 'I am Todd': 30}`
	`>>> confusingDict`
	`{'I am Todd': 30}`

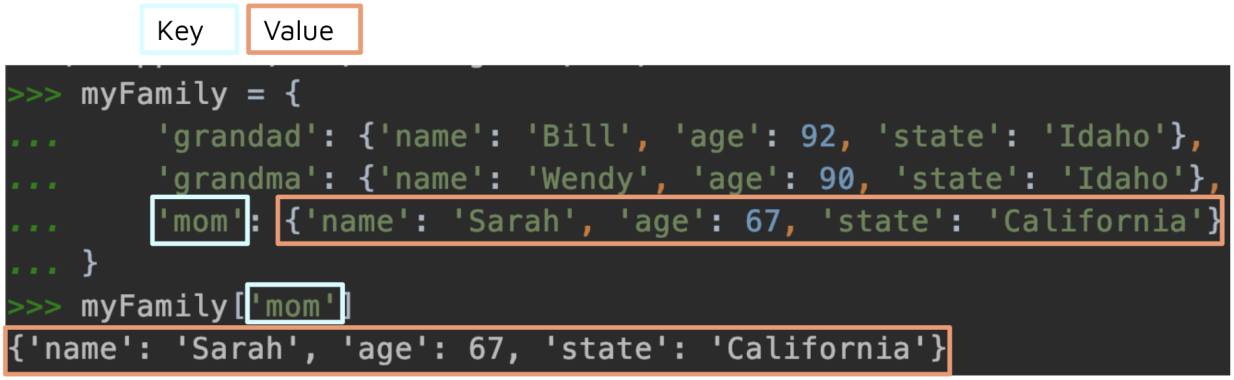
Dictionaries can be nested

Dictionaries can be nested as many times as you want, but accessing the nested items becomes progressively more complicated.

```
myFamily = {
    'grandad': {'name': 'Bill', 'age': 92, 'state': 'Idaho'},
    'grandma': {'name': 'Wendy', 'age': 90, 'state': 'Idaho'},
    'mom': {'name': 'Sarah', 'age': 67, 'state': 'California'}
}
```

Access items in a dictionary

Values in a dictionary are accessed similarly to lists. They use bracket notation, but the key is placed inside of the bracket instead of a list index.

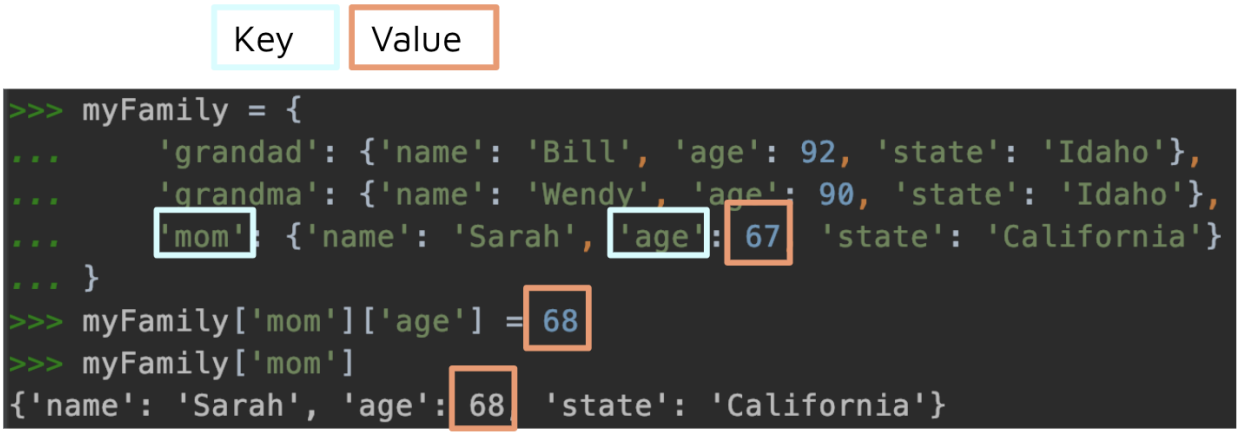


An error occurs if you use a key that doesn't exist in the dictionary.

```
>>> myFamily = {
...     'grandad': {'name': 'Bill', 'age': 92, 'state': 'Idaho'},
...     'grandma': {'name': 'Wendy', 'age': 90, 'state': 'Idaho'},
...     'mom': {'name': 'Sarah', 'age': 67, 'state': 'California'}
... }
>>> myFamily['dad']
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'dad'
```

Update items in a dictionary

To update a dictionary item, you access it and assign it a new value.



Add items in a dictionary

To add a new item to a dictionary, use the same bracket notation to access an existing value, but use a new key instead of an existing one and assign it a value. By definition, the new key doesn't have to exist when the dictionary is first created.

```
>>> myFamily = {
...     'grandad': {'name': 'Bill', 'age': 92, 'state': 'Idaho'},
...     'grandma': {'name': 'Wendy', 'age': 90, 'state': 'Idaho'},
...     'mom': {'name': 'Sarah', 'age': 67, 'state': 'California'}
... }
>>> myFamily['dad'] = {'name': 'James', 'age': 70, 'state': 'California'}
>>> myFamily['dad']
{'name': 'James', 'age': 70, 'state': 'California'}
```

Common dictionary methods and functions

get - (method) allows you to get a dictionary item without knowing whether the key exists.

If you try to use a key that doesn't exist with the normal bracket notation, you will get an error.

```
>>> myFruits = {'apples': 2, 'bananas': 4, 'pineapples': 1}
>>> myFruits['tomato']
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'tomato'
```

With the "get" method, you will get None instead of an error.

```
>>> myFruits = {'apples': 2, 'bananas': 4, 'pineapples': 1}
>>> myFruits.get('tomato')
```

Optionally, you can specify a default value to get back if the key doesn't exist.

```
>>> myFruits = {'apples': 2, 'bananas': 4, 'pineapples': 1}
>>> myFruits.get('apples', 'This is not in my fruits')
2
>>> myFruits.get('tomato', 'This is not in my fruits')
'This is not in my fruits'
```

items - (method) returns a list of key, value pairs as tuples

```
>>> myFruits = {'apples': 2, 'bananas': 4, 'pineapples': 1}
>>> list(myFruits.items())
[('apples', 2), ('bananas', 4), ('pineapples', 1)]
```

values - (method) returns a list of dictionary values

```
>>> myFruits = {'apples': 2, 'bananas': 4, 'pineapples': 1}
>>> list(myFruits.values())
[2, 4, 1]
```

keys - (method) returns a list of dictionary keys

```
>>> myFruits = {'apples': 2, 'bananas': 4, 'pineapples': 1}
>>> list(myFruits.keys())
['apples', 'bananas', 'pineapples']
```

pop - (method) Remove a key, value pair from a dictionary and return the value

```
>>> myFruits = {'apples': 2, 'bananas': 4, 'pineapples': 1}
>>> myFruits.pop('bananas')
4
>>> myFruits
{'apples': 2, 'pineapples': 1}
```

del - (function) Remove a key, value pair from a dictionary. Doesn't return a value.

```
>>> myFruits = {'apples': 2, 'bananas': 4, 'pineapples': 1}
>>> del myFruits['bananas']
>>> myFruits
{'apples': 2, 'pineapples': 1}
```

Check if a key is in a dictionary

Use the "in" word to check if a key is in a dictionary.

```
>>> myFruits = {'apples': 2, 'bananas': 4, 'pineapples': 1}
>>> if 'tomato' in myFruits:
...     print('Hey a tomato!')
... else:
...     print('Tomatoes are not real fruits')
...
Tomatoes are not real fruits
```

[Advanced] Dictionary comprehension

Similar to list comprehension, dictionary comprehension allows you to do the following in a single line:

- Create a new dictionary
- Loop through existing lists and/or dictionaries
- Use conditional(s)
- Add to the new dictionary

```
>>> myFruits = {'apples': 2, 'bananas': 4, 'pineapples': 1}
>>> approvedFruits = ['apples', 'bananas']
>>> myApprovedFruits = {fruit: quantity for fruit, quantity in myFruits.items() if fruit in approvedFruits}
>>> myApprovedFruits
{'apples': 2, 'bananas': 4}
```

This can be expressed in the longer way, like this:

```
>>> myFruits = {'apples': 2, 'bananas': 4, 'pineapples': 1}
>>> approvedFruits = ['apples', 'bananas']
>>> myApprovedFruits = {}
>>> for fruit, quantity in myFruits.items():
...     if fruit in approvedFruits:
...         myApprovedFruits[fruit] = quantity
...
>>> myApprovedFruits
{'apples': 2, 'bananas': 4}
```

[Advanced] defaultdict function

The defaultdict function must be imported. It is a useful way to create default values when you're unsure if a key already exists in a dictionary.

Commentary	**Python console**
defaultdict must be imported	`>>> from collections import defaultdict`
We create a dictionary "myFruits" with a default of "int". This means that if we try to update a value in the "myFruits" dictionary that doesn't currently exist, it will default to a data type of "int" with a value of 0.	`>>> myFruits = defaultdict(int)`

Neither 'apples' or 'bananas' are in the dictionary so we get the default value of 0 and add the quantity to it.	`>>> for fruit, quantity in [('apples', 3), ('bananas', 2)]:`
	`...     myFruits[fruit] += quantity`
	`...`
	`>>> myFruits`
	`defaultdict(<class 'int'>, {'apples': 3, 'bananas': 2})`

'apples' already exists in the dictionary with a value of 3 so we get that value and add the new quantity to it. 'blueberries' doesn't exist so we get the default of 0 and add to it.	`>>> for fruit, quantity in [('apples', 5), ('blueberries', 7)]:`
	`...     myFruits[fruit] += quantity`
	`...`
	`>>> myFruits`
	`defaultdict(<class 'int'>, {'apples': 8, 'bananas': 2, 'blueberries': 7})`

Sets

Sets are similar to lists in a few ways. They are both:

- Iterable
- Mutable

- Contain comma separated values

One good reason to use a set instead of a list is because sets ensure the same value can only appear once. Here are the differences between sets and lists:

Sets	**Lists**
Enforces unique values	The same value can appear multiple times
Created with braces ({})	Created with brackets ([])
Can't access values directly	Values accessed using indexes
Items are unordered	Items are ordered
Items in a set cannot be mutable	Items in a list can be mutable

In addition to the above differences, the methods for sets are different than those for lists.

Create a set

Sets can be created in two ways:

```
>>> set([1, 2, 3])
{1, 2, 3}
>>> {1, 2, 3}
{1, 2, 3}
```

Notice that the second way of creating a set is similar to how dictionaries are created. The difference is that only values are provided instead of key/value pairs.

Items in a set are unique

While sets are not as good as lists for accessing data, they are a great way to get rid of duplicates.

```
>>> myDuplicateList = [1, 'a', 2, 'y', 'y', 1, 4, 3, 'b', 'a', 2, 'b']
>>> mySet = set(myDuplicateList)
>>> mySet
{1, 2, 3, 4, 'b', 'y', 'a'}
```

You can convert a list to a set to get rid of duplicates and then convert back to a list. Note the example below uses a list comprehension.

```
>>> myDuplicateList = [1, 'a', 2, 'y', 'y', 1, 4, 3, 'b', 'a', 2, 'b']
>>> myUniqueList = [x for x in set(myDuplicateList)]
>>> myUniqueList
[1, 2, 3, 4, 'b', 'y', 'a']
```

Items in a set cannot be mutable

```
>>> mySet = {1, 2, 3}
>>> myMutableList = [1, 4, 2, 4]
>>> mySet = {1, 2, myMutableList}
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Common set methods

add - adds a new item to a set

```
>>> mySet = {1, 2, 3}
>>> mySet.add(4)
>>> mySet
{1, 2, 3, 4}
```

update - adds a set to the existing one

```
>>> mySet = {1, 2, 3}
>>> mySet.update({4, 5, 4, 6, 6, 6})
>>> mySet
{1, 2, 3, 4, 5, 6}
```

discard - removes an item from a set. Will not raise an error if the item isn't in the set.

```
>>> mySet = {1, 2, 3}
>>> mySet.discard(2)
>>> mySet
{1, 3}
>>> mySet.discard(5)
>>> mySet
{1, 3}
```

remove - removes an item from a set. Will raise an error if the item isn't in the set.

```
>>> mySet = {1, 2, 3}
>>> mySet.remove(2)
>>> mySet
{1, 3}
>>> mySet.remove(5)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 5
```

pop - removes an item from the set and returns the value.

```
>>> mySet = {1, 2, 3}
>>> myNum = mySet.pop()
>>> myNum
1
```

Note: Sets also have a number of methods which support set comparisons. This is beyond the scope of this document.

Check if a value is in a set

This is accomplished the same way as it is for lists, using the "in" keyword.

```
>>> mySet = {'apples', 'oranges', 'bananas'}
>>> 'oranges' in mySet
True
>>> 'berries' in mySet
False
```

Loop through items in a set

This is accomplished the same way as it is for lists, using the "for" loop:

```
>>> mySet = {'apples', 'oranges', 'bananas'}
>>> for fruit in mySet:
...     print(fruit)
...
bananas
oranges
apples
```

Tuples

A tuple is an immutable list, meaning it can't be changed once created. Tuples are created using a value, at least one comma and optional parenthesis.

```
>>> myTuple = (1, 2, 3)
```

The parenthesis are not actually required.

```
>>> myTuple = 1, 2, 3
>>> myTuple
(1, 2, 3)
```

You need at least one comma so Python knows this is a tuple

```
>>> thisIsNotATuple = 1
>>> type(thisIsNotATuple)
<class 'int'>
>>> thisIsNotATuple = (1)
>>> type(thisIsNotATuple)
<class 'int'>
>>> thisIsATuple = 1,
>>> type(thisIsATuple)
<class 'tuple'>
>>> thisIsATuple = (1,)
>>> type(thisIsATuple)
<class 'tuple'>
```

Just like lists, tuples are a type of iterable, making them ideal in a "for" loop.

```
>>> myTodos = ('coffee', 'read news', 'work', 'workout', 'eat')
>>> for todo in myTodos:
...     print(f'I will {todo} today')
...
I will coffee today
I will read news today
I will work today
I will workout today
I will eat today
```

If it turns out that you need some of the mutable properties of a list, you can always convert a tuple to a list.

```
>>> myTodos = ('coffee', 'read news', 'work', 'workout', 'eat')
>>> myTodos = list(myTodos)
>>> myTodos
['coffee', 'read news', 'work', 'workout', 'eat']
>>> myTodos.append('sleep')
>>> myTodos
['coffee', 'read news', 'work', 'workout', 'eat', 'sleep']
```

Access items in a tuple

Items in a tuple are accessed the same way they are in a list - using the index of the item you want to retrieve. Remember that all iterables in Python start with an index of 0.

```
>>> myTodos = ('coffee', 'read news', 'work', 'workout', 'eat')
>>> myTodos[1]
'read news'
```

Unlike lists, when multiple items are accessed in a tuple, the value returned is a tuple with the subset of values.

```
>>> myTodos[:2]
('coffee', 'read news')
```

Strings

Strings are 0 or more characters surrounded by double or single quotes.

```
>>> thisIsAString = 'string cheese'
>>> thisIsAlsoAString = "string cheese"
```

It doesn't matter if you use double or single quotes, but the opening and closing quotes must match.

```
>>> notAllowed = 'abc"
```

Strings can be accessed similar to lists

Characters within a string can be accessed the same way you would access a list or tuple.

```
>>> myWords = ['H', 'e', 'l', 'l', 'o', ' ', 'f', 'r', 'i', 'e', 'n', 'd']
>>> myStringWords = 'Hello friend'
>>> myWords[1:3]
['e', 'l']
>>> myStringWords[1:3]
'el'
```

You can even loop through the characters in a string, as you would a list or tuple.

```
>>> for char in 'Todd':
...     print(char)
...
T
o
d
d
```

However strings are immutable so you can't modify them like you would a list.

```
>>> myStringWords = 'Hello friend'
>>> del myStringWords[4]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'str' object doesn't support item deletion
```

String concatenation

Strings can be combined in a few different ways.

They can be added together using the "+" sign.

```
>>> myWords = 'Hello' + ' ' + 'friend'
>>> myWords
'Hello friend'
```

They can be joined together using the "join" method.

This is inserted between each consecutive word

```
>>> myWords = ['Hello', 'my', 'only', 'friend']
>>> ' '.join(myWords)
'Hello my only friend'
```

```
>>> 'scooby doo'.join(myWords)
'Helloscooby doomyscooby dooonlyscooby doofriend'
```

They can be inserted using placeholders. This is the "old school" way to do it which I wouldn't recommend. Each substring you want to insert is represented by "%s" (the "s" indicates the format to use and we won't go into other formats). The string is followed by the "%" sign which is followed by a tuple of strings to insert in place of the "%s"s.

```
>>> 'Hello %s nice to meet you. My name is %s' % ('Scooby', 'Todd')
'Hello Scooby nice to meet you. My name is Todd'
```

They can be inserted using "f" strings. This is the "new school" way to do it which I would recommend. The letter "f" is placed at the beginning of the string and any variable you wish to insert is surrounded by braces.

```
>>> myName = 'Todd'
>>> dogName = 'Scooby'
>>> f'Hello {dogName} nice to meet you. My name is {myName}'
'Hello Scooby nice to meet you. My name is Todd'
```

You can also compute values within f strings.

```
>>> f'Hi my name is {myName} and I am {myAgeInMonths / 12} years old'
'Hi my name is Todd and I am 72.66666666666667 years old'
```

Escaping characters

Certain characters or groups of characters contain special meaning in Python. For example, the single quote can be used as a contraction (e.g. don't). This is problematic when you are using single quotes to indicate the start and end of a string.

```
>>> myWord = 'don't do this'
```

One way around this problem is to use double quotes when you know you need to use a single quote in the string.

```
>>> myWord = "don't do this"
```

Instead, however, you can use the backslash character to "escape" a character. "Escaping" means that you are indicating that the character after the backslash should be treated as a string and should not be used to mean anything special.

```
>>> myWord = 'don\'t do this'
```

Another issue that can occur, especially when using file paths, is that the backslash character followed by a certain letter can have special meaning.

```
>>> thisIsSpecial = '\b'
>>> thisIsNotSpecial = '\c'
```

```
>>> thisFilePathWillNotWork = '\backgrounds\mountains\reno'
```

Specifically because of this, Python provides "r" strings (standing for raw strings) which will treat all backslashes as normal.

```
>>> thisFilePathWillWork = r'\backgrounds\mountains\reno'
```

[Advanced] Regular expressions

Regular expressions are an extremely powerful tool to use pattern matching with strings. This can be helpful for all sorts of reasons including scrubbing and cleaning messy data. I highly recommend learning regular expressions, but they will not be covered in detail here. There is plenty of documentation to explain how to use them - just Google "Python regular expression". Below are some brief examples of how they can be used to demonstrate their utility.

```
>>> import re
>>> phoneNumbers = ['cat29', '303-891-3421', '(719)-201-9981', '416-']
>>> phoneNumberPattern = '[0-9()]{3,5}[0-9\-]{9}'
>>> realPhoneNumbers = [phoneNumber for phoneNumber in phoneNumbers if re.match(phoneNumberPattern, phoneNumber)]
>>> realPhoneNumbers
['303-891-3421', '(719)-201-9981']
```

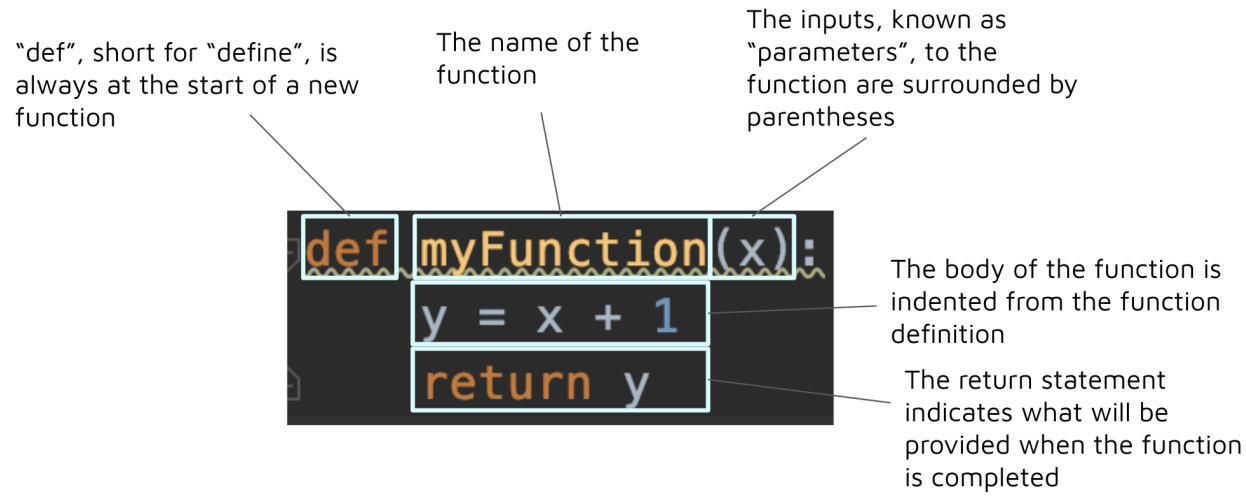
```
>>> import re
>>> mySentence = 'I\'d like to find the 29 number in this sentence.'
>>> numberMatcher = '.*?(?P<myNumber>[0-9]+).*'
>>> matched = re.match(numberMatcher, mySentence)
>>> matched.group('myNumber')
'29'
```

Functions

Functions allow you to define a process which can be re-used and which results in the same output when given the same input(s). For example, a recipe in a cookbook is a function. It defines the ingredients (inputs), the cooking instructions (the process), and the final meal (the output). "Don't repeat yourself" (D.R.Y.) is a common principle in coding. Functions (and loops) make that possible.

Create a function

Functions are created using the following syntax:



To continue with the recipe analogy, here is an example of an overly simplified cake recipe

```
def bakeCake(sugar, flower, butter, bakingSoda):
    mixedIngredients = sugar + flower + butter + bakingSoda
    ovenBake = 350
    return 'cake'
```

Functions do not require any arguments:

```
>>> def noArgFunction():
...     return 2 + 4
...
>>> noArgFunction()
6
```

Call a function

When a function is defined, the inner body of the function is not actually run. The function only runs when it is "called". Defining a function is like writing a new recipe for a cake. When you write the recipe, you don't actually bake the cake. When you follow (call) the recipe, then the end result is the actual cake. Functions can be called by using the function name, followed by parentheses which contain the argument(s) for the function. Functions can be called as many times as you want (that's the point of functions).

```
>>> def myFunction(x):
...     y = x + 1
...     return y
...
>>> myFunction(1)
2
>>> myFunction(5)
6
>>> myFunction(108)
109
```

Return statement

Most often, a function will return a value, but it is not required. A common reason a function may not return a value is that it is modifying mutable object(s). For example:

```
>>> myAccount = {'ownerName': 'Todd', 'accountValue': 500}
>>> yourAccount = {'ownerName': 'Maddie', 'accountValue': 1200}
>>> def transferMoney(transferFromAccount, transferToAccount, amountToTransfer):
...     if transferFromAccount['accountValue'] < amountToTransfer:
...         print('Not enough money in account to complete transaction')
...     else:
...         transferFromAccount['accountValue'] -= amountToTransfer
...         transferToAccount['accountValue'] += amountToTransfer
...         print(f'Transferred ${amountToTransfer} from {transferFromAccount["ownerName"]} to {transferToAccount["ownerName"]}')
...
>>> transferMoney(myAccount, yourAccount, 70)
Transferred $70 from Todd to Maddie
>>> myAccount
{'ownerName': 'Todd', 'accountValue': 430}
>>> yourAccount
{'ownerName': 'Maddie', 'accountValue': 1270}
```

"myAccount" and "yourAccount" are both dictionaries which are mutable objects. The "transferMoney" function modifies both of the dictionaries, but does not return any value.

Most of the time, you will want to return a value. For example:

```
>>> myBank = {'ownerName': 'Todd', 'accounts': [230, 34, 608]}
>>> def getTotalWorth(bankAccount):
...     totalWorth = 0
...     for accountVal in bankAccount['accounts']:
...         totalWorth += accountVal
...
...     return totalWorth
...
>>> myWorth = getTotalWorth(myBank)
>>> myWorth
872
```

The "return" statement automatically exits the function. It is possible to have as many return statements as you want in a function, but only one of the return statements will ever be executed for each function call.

```
>>> def isTimeForDinner(currentHour, dinnerHour):
...     if currentHour >= dinnerHour:
...         print('Start the oven')
...         return True
...     print('I\'m hungry')
...     return False
...
>>> isTimeForDinner(20, 18)
Start the oven
True
>>> isTimeForDinner(20, 21)
I'm hungry
False
```

Functions can call other functions

If you think about coding like legos, you can see that you can start with simple building blocks and create structures of increasing complexity. Similarly, you can create functions (blocks of code) which can be combined in flexible ways to create programs of increasing complexity.

The "getLoanApproval" function use two other functions within it - "getTotalWorth" and "print"

```
>>> myBank = {'ownerName': 'Todd', 'accounts': [230, 34, 608]}
>>> def getTotalWorth(bankAccount):
...     totalWorth = 0
...     for accountVal in bankAccount['accounts']:
...         totalWorth += accountVal
...
...     return totalWorth
...
>>> def getLoanApproval(bankAccount, approvalLimit):
...     totalWorth = getTotalWorth(bankAccount)
...     if totalWorth >= approvalLimit:
...         print('You\'re approved!')
...         return True
...     else:
...         print('Sorry you\'re declined.')
...         return False
...
>>> getLoanApproval(myBank, 1000)
Sorry you're declined.
False
>>> isApproved = getLoanApproval(myBank, 500)
You're approved!
>>> isApproved
True
```

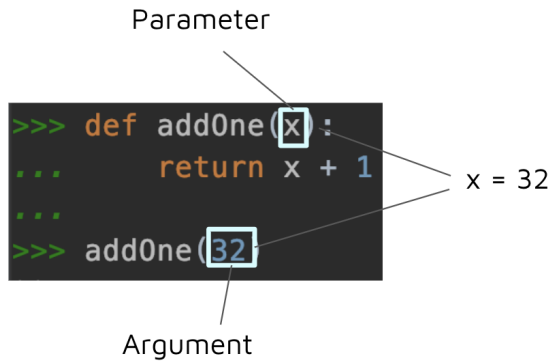
Functions can be defined within other functions

Most of the time, it doesn't make sense to "nest" a function within another function, but there are some reasons to do it based on the scope of certain variables. We will only make note that it is possible to nest functions, but you should stop and question whether it is necessary to do so if you ever nest a function in your code.

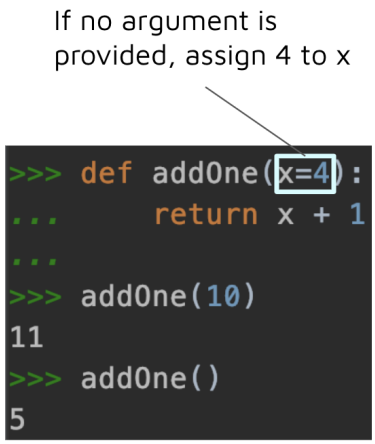
```
>>> def addOne(x):
...     def addTwo(y):
...         return y + 2
...     return addTwo(x) + 1
...
>>> addOne(3)
6
```

Parameters and arguments

Parameters define the inputs to the function. A parameter is the variable name which will be assigned to a value when the function is called. The value passed to the function is called the argument.

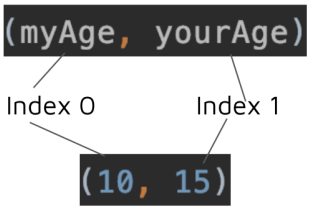


Parameters can have default values.



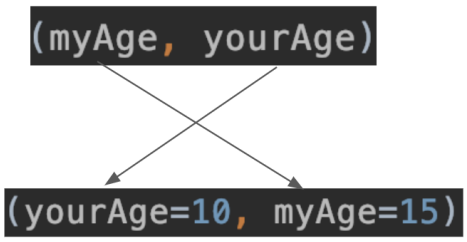
When just arguments are used to call the function, the index of the argument is used to determine which parameter it is assigned to. These are known as "positional" arguments.

```
>>> def areYouOlder(myAge, yourAge):
...     return myAge < yourAge
...
>>> areYouOlder(10, 15)
True
```



The name(s) of the parameter(s) can be used instead. When the names are used, the index is ignored and instead the parameter name is used as a key to get the corresponding argument value. These are known as "keyword" arguments.

```
>>> def areYouOlder(myAge, yourAge):
...     return myAge < yourAge
...
>>> areYouOlder(yourAge=10, myAge=15)
False
```



A mixture of positional and keyword arguments can be used, but the positional arguments must all come before the keyword arguments.

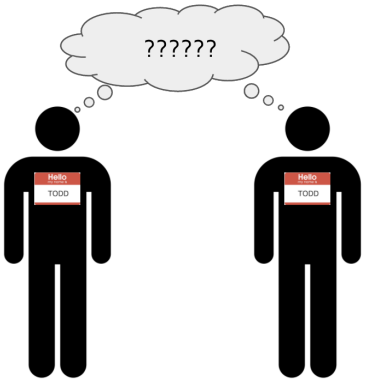
```
>>> def areYouTallEnough(heightInches, isRollerCoaster=True, isParentWithYou=True):
...     if isParentWithYou:
...         return True
...     if isRollerCoaster:
...         return heightInches > 60
...     return heightInches > 48
...
>>> areYouTallEnough(50, isParentWithYou=False)
False
>>> areYouTallEnough(50, isRollerCoaster=False)
True
>>> areYouTallEnough(isRollerCoaster=False, 50)
  File "<input>", line 1
SyntaxError: positional argument follows keyword argument
```

Scope

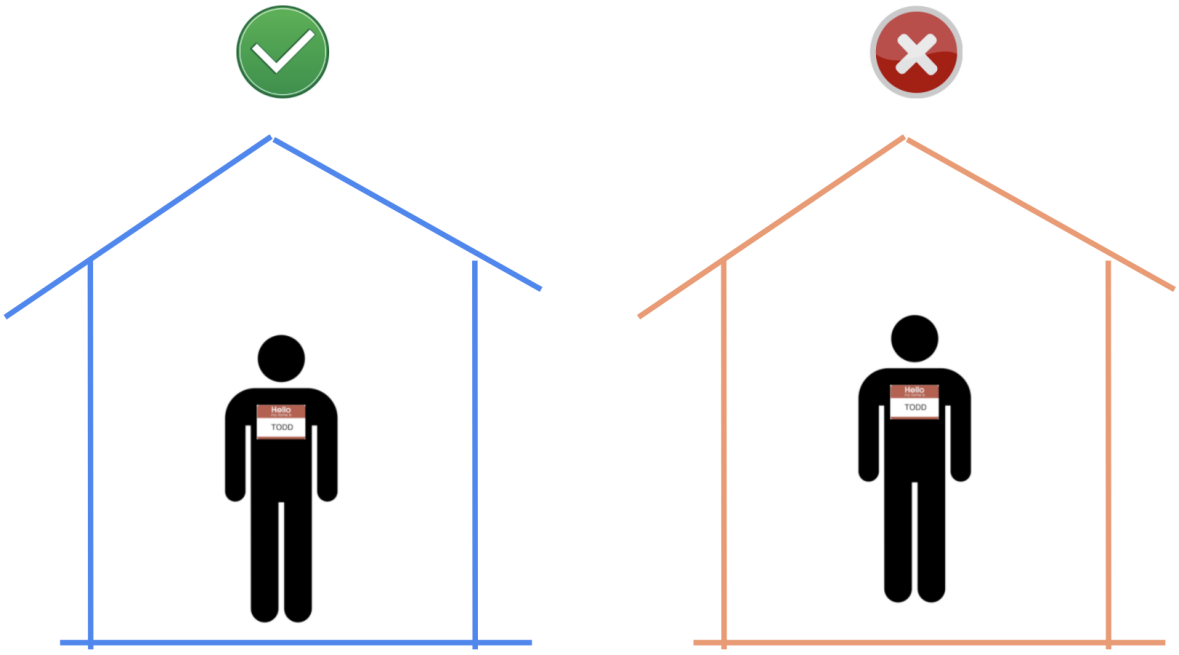
Scope is a very important concept to understand. Scope determines the context in which a statement is executed. Said another way, scope determines when a variable can be found and used at a certain point in the code. This is important for a couple reasons:

- We want to avoid ambiguity in what a variable refers to
- Not all parts of the code should, or need, to have "knowledge" of other parts of the code

In the section on variables, we discussed that variables must be unique. This is true, but a more accurate statement is that variables within the same scope must be unique. If someone yells "Todd" and there are two Todd's, that would cause confusion:



But if the two Todds live in different houses (scopes) and someone yells "Todd in the blue house", then there is no longer confusion.



Also think about two employees at a company. One employee is in marketing and the other is in finance. Both have individual knowledge that (hopefully) allows them to do their jobs well,

but the marketing employee is not expected to know about finance or vice versa. However, the marketing employee can ask for pieces of information from the finance employee that might be required to do their job.

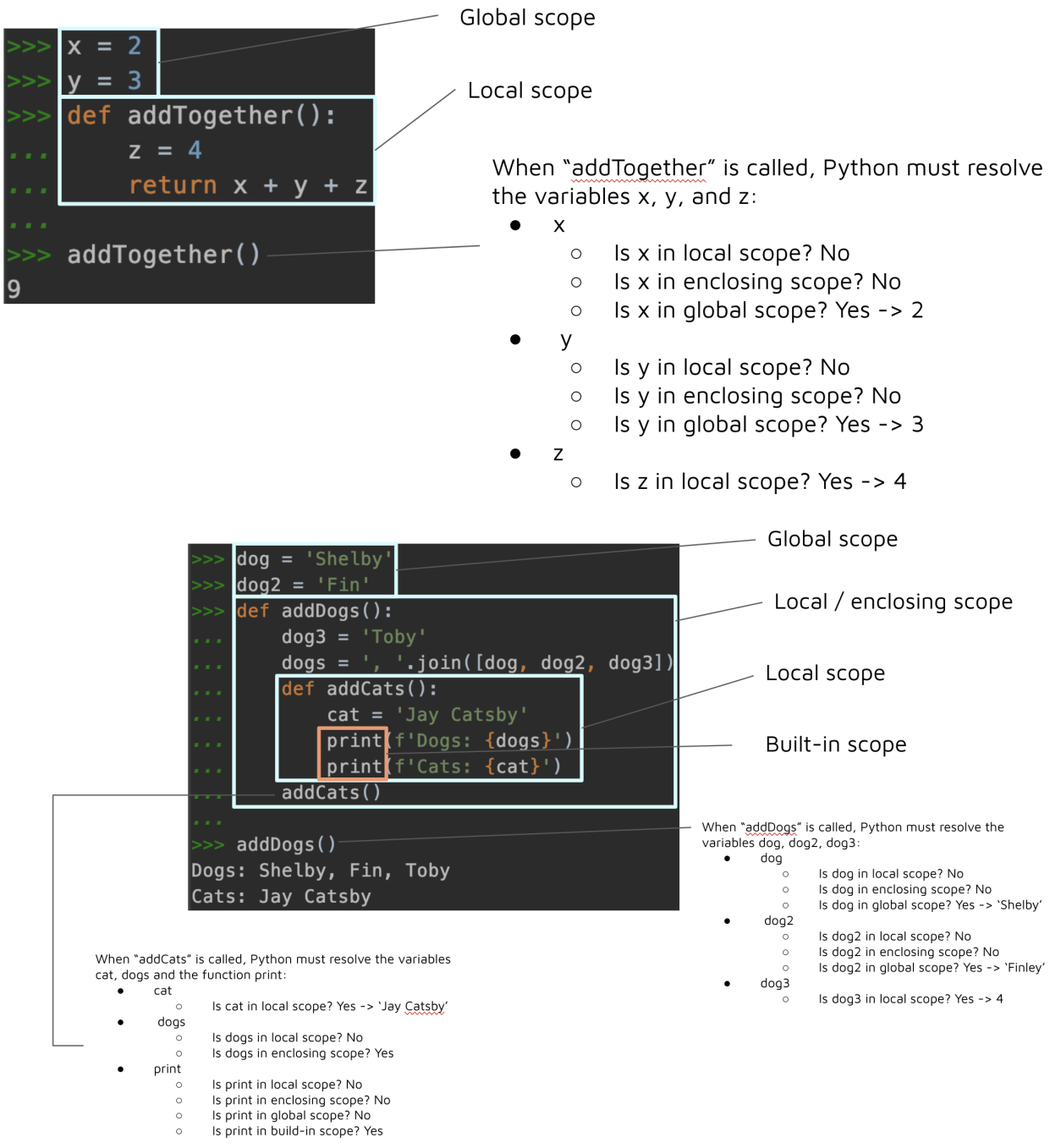
In Python, scope is determined by the LEGB rule. Variables, functions, and classes are all examples of "objects" that we may need to search for in our scope.

- **L**ocal (function) scope - objects within a function are "local" to that function and cannot be accessed outside of the function.
- **E**nclosing scope - objects defined within functions which are defined within another function can be accessed by the enclosing (outer) function.
- **G**lobal scope - objects defined within a file or module are accessible throughout the file or module.
- **B**uilt-in scope - certain objects are automatically accessible anywhere. These are the most common "tools" used in Python.

Python will start searching for the object at the local level, if it's not found there, it will search at the enclosing level, and so on all the way up to the built-in scope. Roughly speaking, the below example shows how scope search works:

```
>>> localObjects = {'this': 1, 'that': 2}
>>> enclosingObjects = {'this': 3, 'other': 4}
>>> globalObjects = {'the': 5, 'other': 6}
>>> builtInObjects = {'print': print, 'len': len}
>>> def scopeSearch(objectName):
...     if objectName in localObjects:
...         return localObjects[objectName]
...     if objectName in enclosingObjects:
...         return enclosingObjects[objectName]
...     if objectName in globalObjects:
...         return globalObjects[objectName]
...     if objectName in builtInObjects:
...         return builtInObjects[objectName]
...     return f'Error: No object named {objectName} found'
...
>>> scopeSearch('this')
1
>>> scopeSearch('other')
4
>>> scopeSearch('len')
<built-in function len>
>>> scopeSearch('something')
'Error: No object named something found'
```

Here are some examples of how scope is determined:



Functions vs. methods

A method is a special type of function that is owned by a class. The distinction will become apparent once you understand classes, but for now, we can still draw differences at a high level. The main difference is that methods are accessed through a class using "dot notation". You have already seen this in examples from previous sections.

"append" is a method accessible by any list

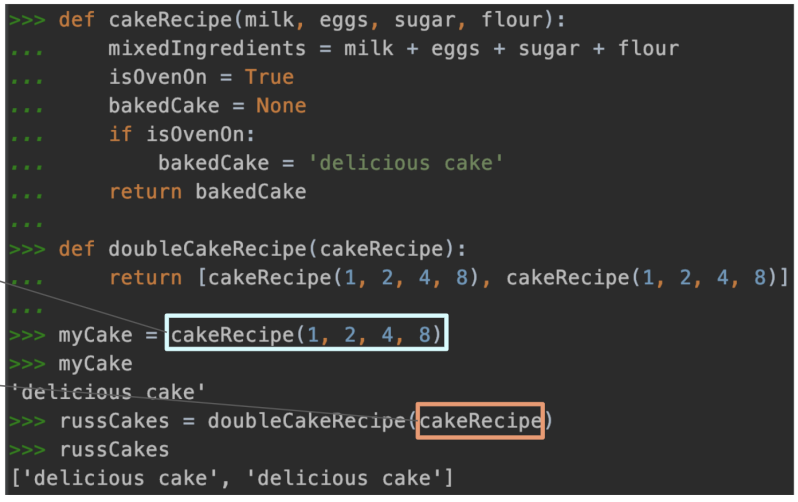
"len" is a function that can be used on many data types

Functions are objects

The fact that functions (and everything in Python) are objects is extremely powerful. This will make more sense when you understand Object Oriented Programming (OOP). For now, know that because functions are objects, they can be "passed" to other functions just like any other data. Consider my cake recipe analogy. My friend Russ could ask me for a cake in which case, I will follow the recipe myself and hand him the cake (this is equivalent to calling a function). But maybe Russ doesn't want a cake now, but instead just wants the recipe so he can bake a cake (or many cakes!) in the future. I could hand Russ the cake recipe (which is equivalent to passing a function as an object).

To call a function, use the parentheses and pass in the necessary arguments

To refer to the function itself, simply use the function name without parentheses or arguments



When you use parentheses and arguments after the function name, you are instructing Python to "get me the **result** of this function". When you leave off the parentheses and arguments, you are instructing Python "get me this function".

[Advanced] Lambdas

Lambdas are "quick and dirty" functions. They are typically one line functions that aren't worth the extra lines of code to create a normal function. They are created with the syntax "lambda --parameter(s)--: --function body--. With lambda functions, the "return" statement is implied so the term is not actually used. Here is an example:

```
>>> mySimpleFunc = lambda x: x + 1
```

The above example is equivalent to:

```
>>> def mySimpleFunc(x):
...     return x + 1
```

You can use as many parameters as you want:

```
>>> mySimpleFunc = lambda x, y, z: x + y - z
```

Lambdas are called the same way normal functions are called.

```
>>> mySimpleFunc = lambda x, y, z: x + y - z
>>> mySimpleFunc(2, 4, 8)
-2
```

Notice that lambdas are unnamed and don't ever have to be assigned to a variable. This is why they are known as "anonymous functions".

This function cuts off the first letter of a string. Notice that it is never assigned a name

```
>>> myList = ['milk', 'butter', 'juice']
>>> myWeirdList = list(map(lambda listItem: listItem[1:], myList))
>>> myWeirdList
['ilk', 'utter', 'uice']
```

[Advanced] Generators

Computers these days are way better than they were even 5 years ago, not to mention back in the 1980s. Computers have become so powerful that sometimes we can forget that they do have limited memory and processing power. While working with large data sets in Python, you may eventually find the limits of your computer's memory. Generators are a way to manage large amounts of data that can't be handled in memory all at once.

If I ask you to remember 3 random numbers, say 10, 16, 98, chances are you will be able to recall them a few seconds later. These numbers are stored in your short term memory. If I asked you to remember 100 random numbers and recall them a few seconds later, chances are you will not be able to do so. Computers have a short term "working" memory as well and there are limits to what it can hold.

However, if I asked you to remember one number and a simple equation to apply to that number, say --number + 2--, and then asked you to repeatedly apply the equation 100 times, you probably could do that. In this example, if we start with 1, the numbers would be 1, 3, 5, 7, 9 and so on. At any given time, you only need to remember the last number in the sequence and the function to apply to it. That's what generators do. They remember a current state, apply a process, and provide the new state.

In Python, generators are created using the "yield" keyword.

```
>>> def myGenerator(startingNumber):
...     for i in range(10):
...         yield startingNumber
...         startingNumber += 2
...
>>> for num in myGenerator(1):
...     print(num)
...
1
3
5
7
9
11
13
15
17
19
```

The "yield" keyword is like "return", but also instructs the function to "remember where we left off". Here's what happens the first few times the loop runs in the above code:

Loop # in "for num in myGenerator(1)"	startingNumber value	i value
1	1	0
2	3	1
3	5	2

It's important to note that generators can only get the next value (they can't go backwards) and they can only be looped through once. You may also notice that generators are very similar to lists in many ways. In fact, generators can easily be converted into lists.

```
>>> list(myGenerator(1))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Take note that when you convert a generator to a list, you no longer have the benefits that a generator provides - mainly that you have now loaded all of the data into working memory.

To get the next state of a generator, you use the "next" function.

```
>>> newGen = myGenerator(29)
>>> next(newGen)
29
>>> next(newGen)
31
>>> next(newGen)
33
```

When you get to the end of a generator, meaning there are no more values to yield, you will get a Stopiteration exception.

```
>>> next(newGen)
45
>>> next(newGen)
47
>>> next(newGen)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
```

Why didn't we get this exception when we used the "for" loop in the above example? Well that's actually what for loops do. They call the next function until they hit the "StopIteration" error and instead of raising the error, they exit the loop. You can learn more about error handling in this section. You can actually see that any iterable, not just generators, can be iterated over in the exact same way (using the "next" function).

```
>>> myList = ['apples', 'oranges', 'bananas']
>>> next(myList.__iter__())
'apples'
```

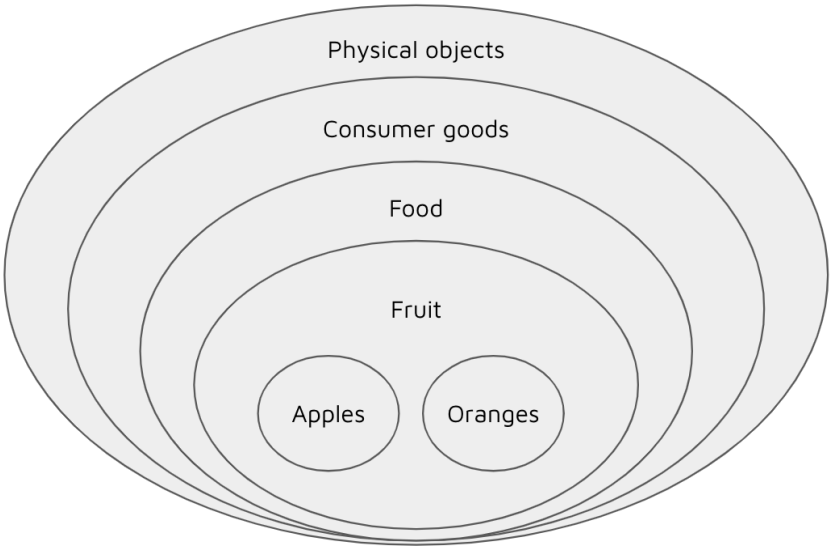
Object Oriented Programming (OOP)

When you first understand the concept of OOP, it's like entering the Matrix. "You're telling me everything is an object?!" Yes everything is an object. Variables are objects, data are objects, functions are objects, and classes are objects. Everything is an object. I think it comes pretty intuitively that variables and data are objects. Functions and classes are where it gets a bit harder to understand. But if you think about functions and classes as paper recipes, it's a bit easier to picture. I can hand you a recipe and you can hand me a recipe. Imagine if there was only one cookbook in the entire world and I owned it. That would be pretty terrible because I would have to cook for everyone (and I don't like to cook). But what if I could make copies of my cookbook and share them with everyone else. Suddenly, we have a lot more cooks! And even better, each cook can modify the cookbook to improve existing recipes and create new recipes. This is the power of OOP!

Object inheritance

What do you get when you add 3 apples and 4 oranges? Impossible? No! You have 7 fruits. Everything in life can be grouped in certain ways (classified) based on the intersection of common traits. Apples and oranges look and taste different, but they both have seeds and grow on trees. The more types of items you add to a group, the less that is shared between

them. Apples and oranges can both be classified as food, which can be classified as a consumer good, which can be classified as a physical object.



I certainly wouldn't eat all types of physical objects (rocks for example), but I know that I can pick them up and that they have a certain amount of mass. So thinking about anything that is a physical object is useful to me in some way.

In Python, the broadest group (known as a "class") is the "object" and other "classes" can be sub-classified from there. When a class is part of a broader class, this is known as "inheritance" because the sub-class inherits the attributes and abilities of its "parent" class(es). So for example, the "food" class inherits the ability to be picked up from the "physical objects class" and inherits the ability to be purchased at a store from the "consumer goods" class.

We can verify that everything is, in fact, an object by inspecting the class structure of different objects:

```
>>> import inspect
>>> def getClassStructure(item):
...     return inspect.getmro(type(item))
...
>>> getClassStructure(27)
(<class 'int'>, <class 'object'>)
>>> getClassStructure(True)
(<class 'bool'>, <class 'int'>, <class 'object'>)
>>> getClassStructure(None)
(<class 'NoneType'>, <class 'object'>)
>>> getClassStructure('this sentence')
(<class 'str'>, <class 'object'>)
>>> getClassStructure({'key1': 1, 'key2': 2})
(<class 'dict'>, <class 'object'>)
>>> getClassStructure([1, 2, 3])
(<class 'list'>, <class 'object'>)
>>> def myFunc(x):
...     return x + 1
...
>>> getClassStructure(myFunc)
(<class 'function'>, <class 'object'>)
```

Because the "object" class is the broadest group, there's not much that can be shared (inherited) among all of the sub-classes. The object class only has 3 main abilities:

- Create a new object
- Delete an object
- Compare one object with another

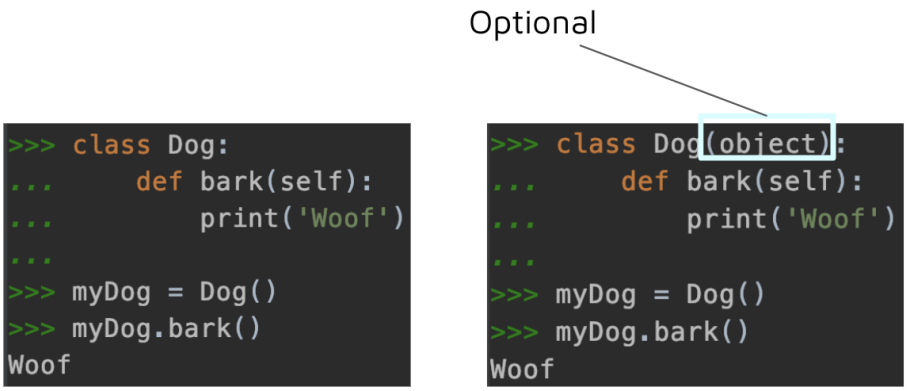
To make objects more useful, sub-classes add new abilities (methods), but don't have to re-implement the abilities that they inherit from their parents. So the "list" class inherits the ability to create, delete, and compare from it's parent, but then implements new abilities like the "append" method to allow users to add new items to a list.

Just like in the fruit example, objects can have a long chain of inheritance. The "bool" class shown above inherits from the "int" class, which inherits from the "object" class. As we talk more about classes, you will see how useful it can be to have multiple levels of inheritance.

Classes

So far we've covered that everything in Python is an object and "object" is the broadest "class" which is then sub-classified into many types of objects like functions, integers, and booleans. We are not limited to the classes defined by Python and can create our own classes. All classes must inherit from the "object" class or any other class (which eventually

inherits from the "object" class). The basic syntax to create a class is "Class --class name--(--parent class--): --class body--". If the class inherits from the "object" class, the parentheses with the parent class is optional.



The above example creates a new "Dog" class which inherits from the "object" class. Keep in mind that, by convention, class names start with an uppercase letter. The Dog class has one method, bark, which prints "Woof" when called.

Once the "Dog" class has been created, we can create instances of "Dog" by using the class name followed by parentheses. The variable "myDog" is an instance of the "Dog" class. Adding the parentheses is equivalent to saying "use the blueprint for a dog to create a specific dog". Because "myDog" is an instance of "Dog", it has the ability (method) to bark. We can call methods using "dot notation", meaning the instance of the class "myDog", followed by a period (dot) ".", followed by the method name, followed by parentheses.

There are a couple of terms used later which need to defined:

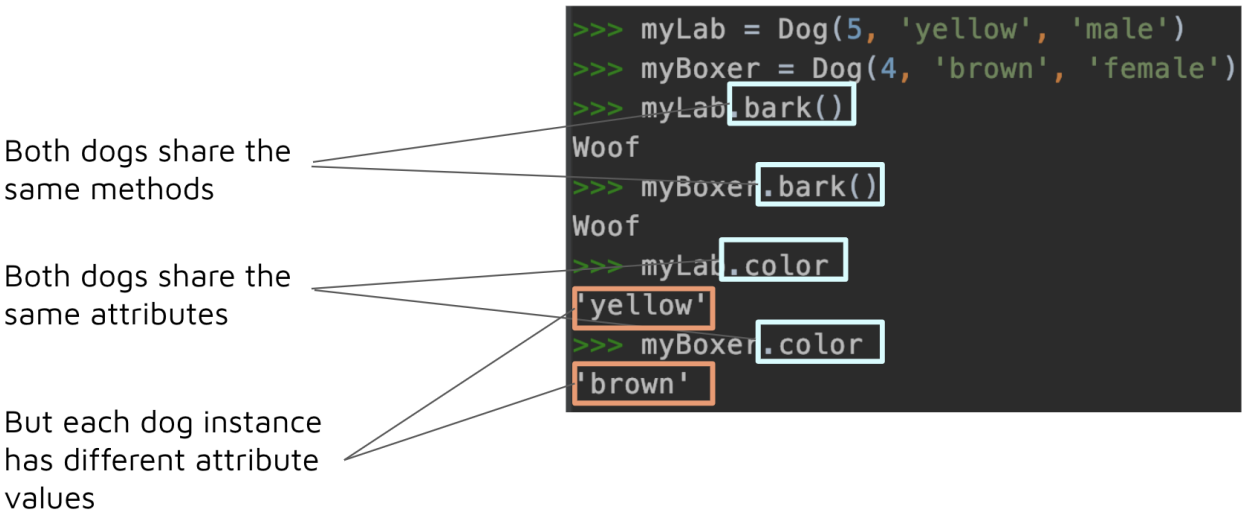
- Attribute - a variable attached to a class
- Method - a function attached to a class

Initialization

A very important concept behind classes is the act of "initialization". Initialization is the act of setting the "initial" state an instance of a class has when it is created. Think about it as being born. Every dog may have the ability to bark, but when a dog is born, it will have a weight, color, sex, etc. which are different from other dogs. So each dog is born (initialized) with common abilities (bark) and attributes (weight, color, sex), but the specific values of the attributes may be different. Using a special method "__init__", short for "initialization", we can set the values of attributes which can be different when a new instance is created.

```
>>> class Dog(object):
...     def __init__(self, weight, color, sex):
...         self.weight = weight
...         self.color = color
...         self.sex = sex
...
...     def bark(self):
...         print('Woof')
...
```

We'll come back to these references to "self" in a bit, but for now, focus on the initialization concept. So the "Dog" class is a blueprint for how to create a dog and the "__init__" method specifically describes what the dog should look like when it is created.



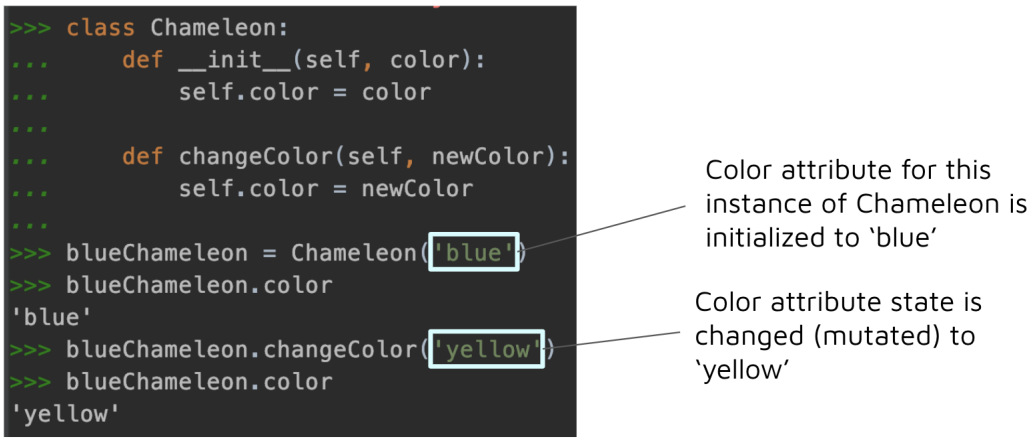
A class is "initialized" using parentheses with the arguments for each parameter within the "__init__" method. Again, ignore the"self" for now - there are three parameters required to initialize a "Dog" instance - weight, color, and sex. When "Dog(5, 'yellow', 'male')" is called, 5 is assigned to weight, 'yellow' is assigned to color, and 'male' is assigned to sex. Then these three arguments are assigned to the instance itself using the "self" term. The values assigned to the instance can then be accessed using "dot notation".

Self

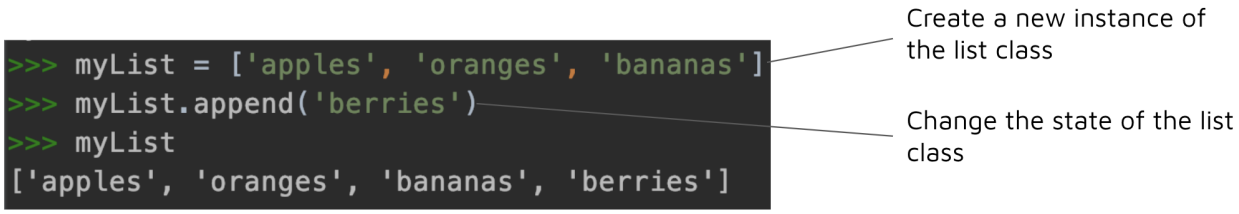
Just like snowflakes, everything in the world is unique if you look hard enough. Even identical twins are different at a cellular level and they can never occupy the same physical space at the exact same time. We've already covered inheritance and the fact that objects can share

common traits that can be classified into groups and the further down the inheritance tree you go, the more different classes share in common (e.g. apples and oranges have a lot in common). Well the "deepest" level of inheritance is the "self". The "self" is an instance of a class. And an instance is different from all other objects, even those that are part of the same class.

Before we explore the self further, it's important to understand that classes are "stateful", meaning they can "remember" the current values of all their attributes and these attributes can be changed. Remember mutation? Mutation is simply changing the state of an object. Let's take a look at a quick example of a state change:



You've already seen state changes many times before:



Coming back to the idea of the "self". In order to know the changed state of an object, we need to know the current state and how to change the current state. The "self" is the current state. A simple way to think about "self" is a dictionary of values that have been set on the instance:

```
>>> class Chameleon:
...     def __init__(self, color):
...         print(f'My current state is: {self.__dict__}')
...         self.color = color
...         print(f'My current state is: {self.__dict__}')
...
...     def changeColor(self, newColor):
...         print(f'My current state is: {self.__dict__}')
...         self.color = newColor
...         print(f'My current state is: {self.__dict__}')
...
>>> purpleChameleon = Chameleon('purple')
My current state is: {}
My current state is: {'color': 'purple'}
>>> purpleChameleon.changeColor('brown')
My current state is: {'color': 'purple'}
My current state is: {'color': 'brown'}
```

This isn't that interesting when you only have one attribute like color, but it becomes way more useful when there are multiple attributes that have to be remembered and shared between methods.

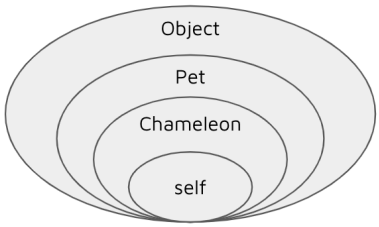
When I first started learning about classes, I wondered why the initialization arguments had to be set on "self" at all. In the example above, why isn't the "color" attribute automatically set to the color argument? The reason is that not all initialization arguments need to be set on the instance and the instance may use a different name for an argument that is passed in. Here's a simple example where that's the case:

```
>>> class Chameleon:
...     def __init__(self, color, ageInMonths):
...         self.mood = 'angry' if color == 'red' else 'happy'
...         self.ageInYears = ageInMonths / 12
...
>>> myPet = Chameleon('red', 23)
>>> myPet.mood
'angry'
>>> myPet.ageInYears
1.9166666666666667
```

Dot notation

Every attribute and method on a class instance can be accessed using "dot notation". You've already seen examples of this, but let's tie this back to inheritance. When "dot notation" is used, Python resolves the name after the dot by searching the inheritance tree from most specific to least specific class.

```
>>> class Pet:
...     def __init__(self, age, weight, food):
...         self.age = age
...         self.weight = weight
...         self.food = food
...
...     def eat(self):
...         print('Nom nom')
...
>>> class Chameleon(Pet):
...     def __init__(self, age, weight, food, color):
...         self.color = color
...         super().__init__(age, weight, food)
...
...     def climb(self):
...         print('Climbing')
...
>>> myPet = Chameleon(3, 23, 'crickets', 'teal')
>>> myPet.climb()
Climbing
>>> myPet.eat()
Nom nom
>>> myPet.fifthLeg = True
>>> myPet.fifthLeg
True
```



- fifthLeg
  - Is it in self? Yes -> True
- climb
  - Is it in self? No
  - Is it in Chameleon? Yes
- eat
  - Is it in self? No
  - Is it in Chameleon? No
  - Is it in Pet? Yes

Note: I don't recommend adding unique attributes directly to "self" like the "fifthLeg" example above. This is only to demonstrate the "deepest" level of inheritance.

Class attributes and methods can be overridden by subclasses. The dot notation resolution described above applies at all times so if an attribute or method is defined at a lower level class, that's the one that will be used. In the example below, the "eat" method of the Chameleon class is found before the "eat" method of the Pet class so that's the one that is used.

```
>>> class Pet:
...     def __init__(self, age, weight, food):
...         self.age = age
...         self.weight = weight
...         self.food = food
...
...     def eat(self):
...         print('Nom nom')
...
>>> class Chameleon(Pet):
...     def __init__(self, age, weight, food, color):
...         self.color = color
...         super().__init__(age, weight, food)
...
...     def climb(self):
...         print('Climbing')
...
...     def eat(self):
...         print('Slurp slurp')
...
>>> myPet = Chameleon(3, 23, 'crickets', 'teal')
>>> myPet.eat()
Slurp slurp
```

Super Method

When methods are overridden by subclasses, the subclass may still want to make use of the original logic of a parent class (also known as the super class). The super method essentially means “get me the class that this class inherits from”.

In the example below, we can see that the __init__ method from Chameleon is run first, but then the super() method calls the __init__ method from Pet.

```
>>> class Pet:
...     def __init__(self, age, weight, food):
...         print('Creating a pet')
...         self.age = age
...         self.weight = weight
...         self.food = food
...
...     def eat(self):
...         print('Nom nom')
...
...
... class Chameleon(Pet):
...     def __init__(self, age, weight, food, color):
...         print('Creating a chameleon')
...         self.color = color
...         super().__init__(age, weight, food)
...
>>> myPet = Chameleon(3, 23, 'crickets', 'teal')
Creating a chameleon
Creating a pet
```

It's not just the __init__ method where the super method can be used. It can be used anywhere in the class.

```
>>> class Pet:
...     def __init__(self, age, weight, food):
...         print('Creating a pet')
...         self.age = age
...         self.weight = weight
...         self.food = food
...
...     def eat(self):
...         print('Nom nom')
...
...
... class Chameleon(Pet):
...     def __init__(self, age, weight, food, color):
...         print('Creating a chameleon')
...         self.color = color
...         super().__init__(age, weight, food)
...
...     def eat(self):
...         print('Slurp slurp')
...         super().eat()
...
>>> myPet = Chameleon(3, 23, 'crickets', 'teal')
Creating a chameleon
Creating a pet
>>> myPet.eat()
Slurp slurp
Nom nom
```

[Advanced] Double under (dunder) methods

Double under (a.k.a. dunder, a.k.a magic) methods are methods surrounded by double unders. The __init__ method is an example of a dunder method. The __init__ dunder method is likely the only type of dunder method you will run into and use, but there are others. The double under essentially means that these are "special" methods. You can see the other double under methods available to an object by using the "__dir__" method.

```
>>> for attribute in myPet.__dir__():
...     if '__' in attribute:
...         print(attribute)
...
__module__
__init__
__doc__
__dict__
__weakref__
__repr__
__hash__
__str__
__getattribute__
__setattr__
__delattr__
__lt__
__le__
__eq__
__ne__
__gt__
__ge__
__new__
__reduce_ex__
__reduce__
__subclasshook__
__init_subclass__
__format__
__sizeof__
__dir__
__class__
```

It may be pretty obvious what some of these dunder methods do. Let's take "__lt__" for example, the "lt" stands for "less than" and this determines the logic that will be used when the less than (<) comparison operator is used. It may be pretty obvious what the logic should be when two integers are compared, but what about when two Chameleon instances are compared?

```
>>> myPet = Chameleon(3, 23, 'crickets', 'teal')
... myOtherPet = Chameleon(2, 40, 'crickets', 'yellow')
>>> myPet < myOtherPet
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: '<' not supported between instances of 'Chameleon' and 'Chameleon'
```

If we do want to compare two Chameleon instances with each other, we have to define the "__lt__" method.

```
>>> class Chameleon(Pet):
...     def __init__(self, age, weight, food, color):
...         self.color = color
...         super().__init__(age, weight, food)
...
...     def eat(self):
...         print('Slurp slurp')
...
...     def __lt__(self, other):
...         return self.weight < other.weight
...
>>> myPet = Chameleon(3, 23, 'crickets', 'teal')
... myOtherPet = Chameleon(2, 40, 'crickets', 'yellow')
>>> myPet < myOtherPet
True
```

[Advanced] Multiple inheritance

It is possible for a class to inherit from multiple parent classes. In practice, I've rarely seen this used and the inheritance structure can get confusing and messy. The below example shows how it works, but I would try to avoid it in most cases.

```
>>> class Pet:
...     def __init__(self, age, weight, food):
...         print('Creating a pet')
...         self.age = age
...         self.weight = weight
...         self.food = food
...
...     def eat(self):
...         print('Nom nom')
...
... class WalkingPet:
...     def walk(self):
...         print('Going for a walk')
...
... class Chameleon(Pet, WalkingPet):
...     def __init__(self, age, weight, food, color):
...         print('Creating a chameleon')
...         self.color = color
...         super().__init__(age, weight, food)
...
...     def eat(self):
...         print('Slurp slurp')
...         super().eat()
...
>>> myPet = Chameleon(3, 23, 'crickets', 'teal')
Creating a chameleon
Creating a pet
>>> myPet.eat()
Slurp slurp
Nom nom
>>> myPet.walk()
Going for a walk
```

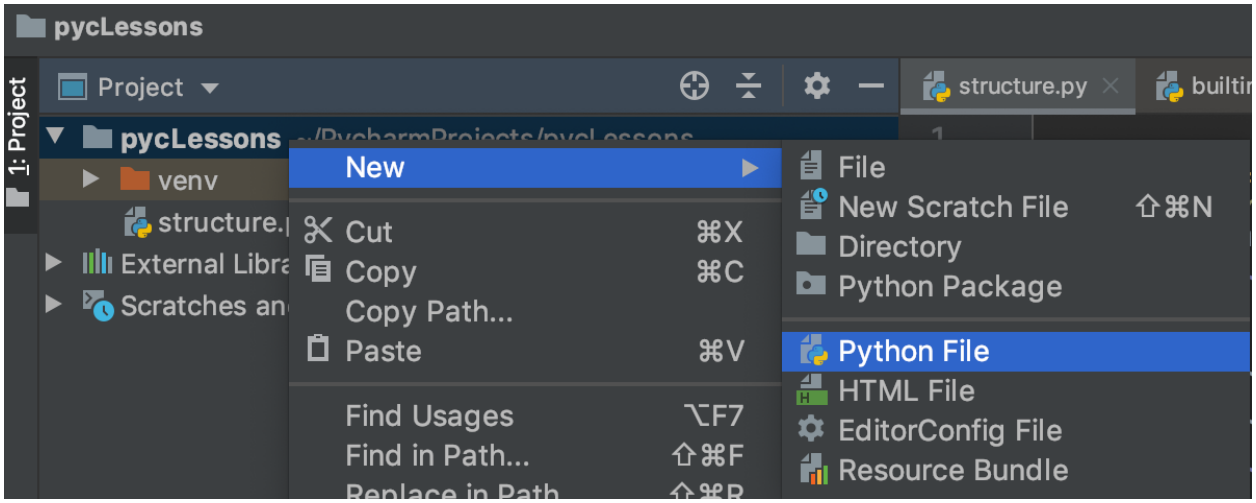
If you do want to have multiple inheritance, you may want to first consider “mixins” as an alternative. Mixins will not be covered in this document, but there are online resources that explain how to use them.

Executing code in a file

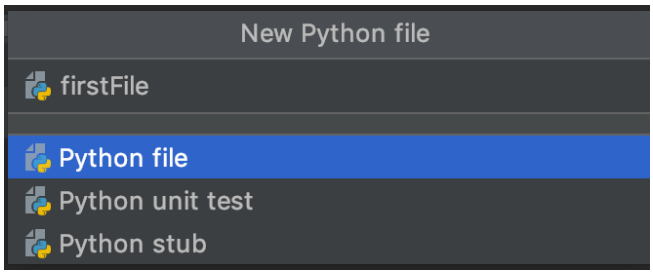
Up to this point, all of the code I have shown was executed in the Python console. Using the console is a great way to quickly show different coding concepts and I often use it to test short snippets of code when I'm not exactly sure how to use a certain function or class. For any serious program, however, we need to be able to save our changes and it would also be nice to have tools like auto-completion, error highlighting, and so on. This is where an integrated development environment (IDE) like PyCharm really shines.

Create a python file

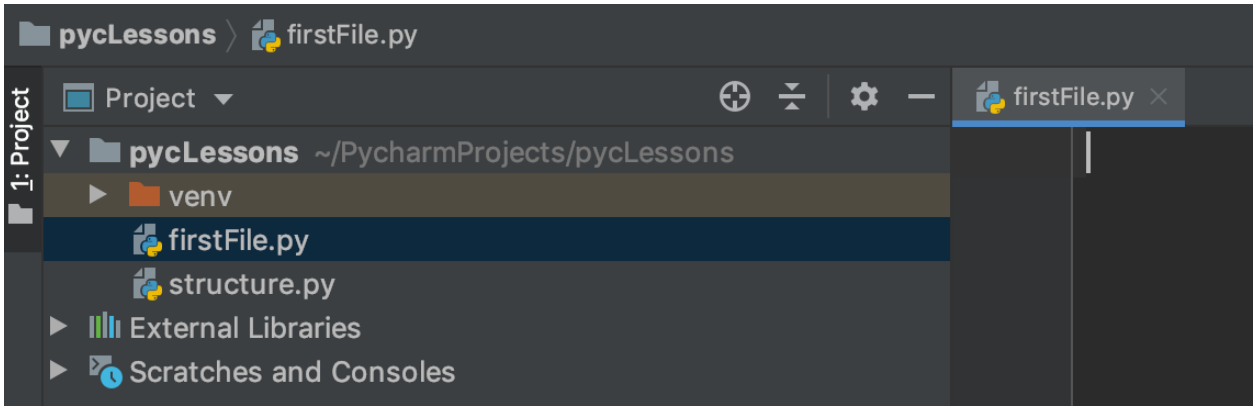
In the project panel of PyCharm (by default in the upper left of the screen), right click on the folder where you want to add a new file. Hover over "New" and then select "Python File" from the dropdown options.



Type the name that you want to assign to the new file (I recommend avoiding spaces in the name) and hit enter (don't change the "Python file" selection).

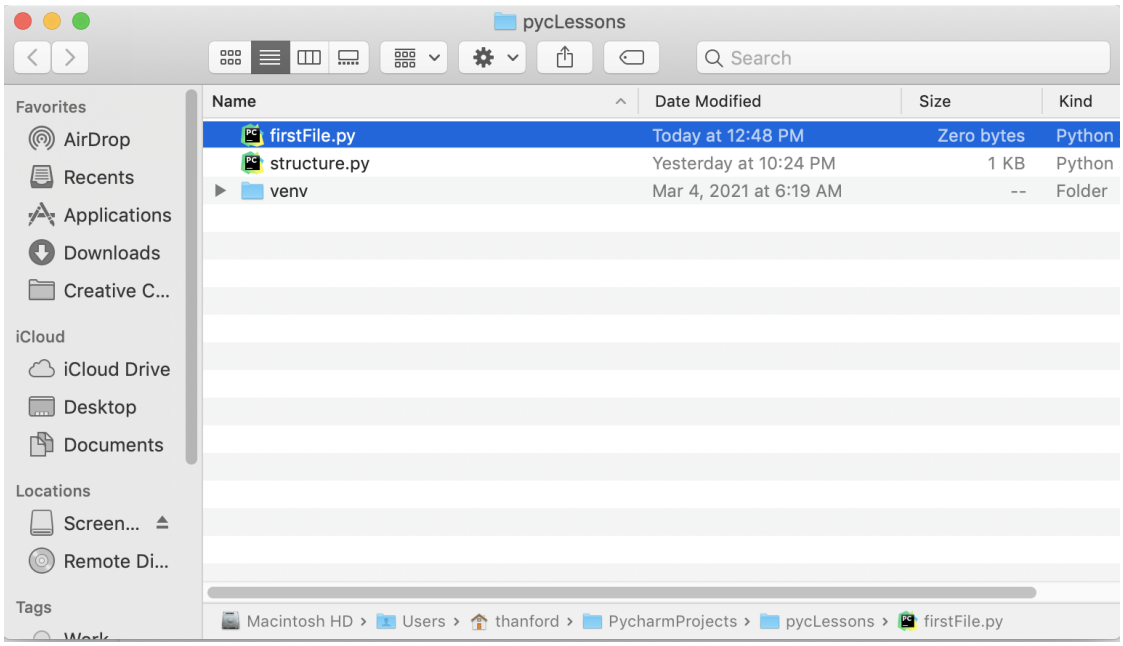


The new file will automatically open a new tab in the main PyCharm window.

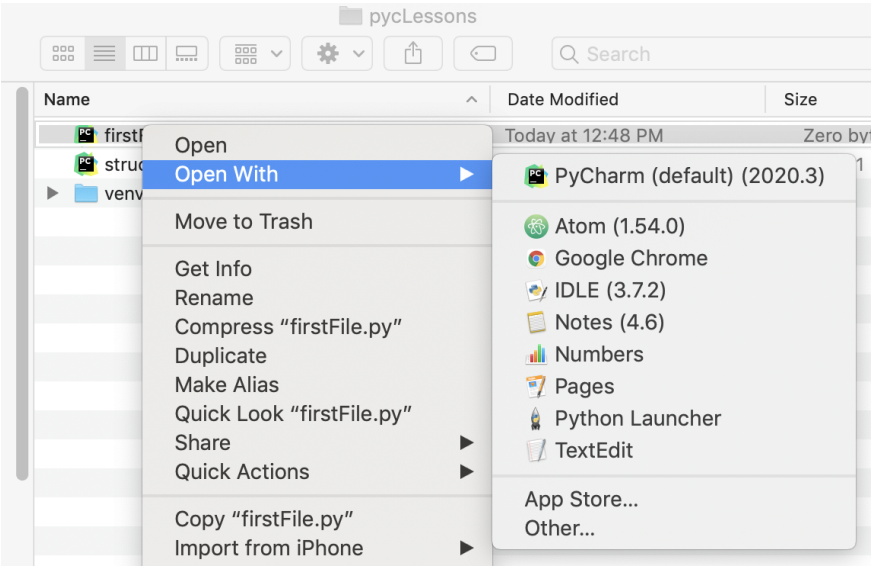


Keep in mind that there's nothing special about PyCharm file creation. PyCharm is really just a fancy text editor that sits on top of files on your computer.

This new file is saved in a folder on my computer:



And I can open it using any program I want:



Edit a python file

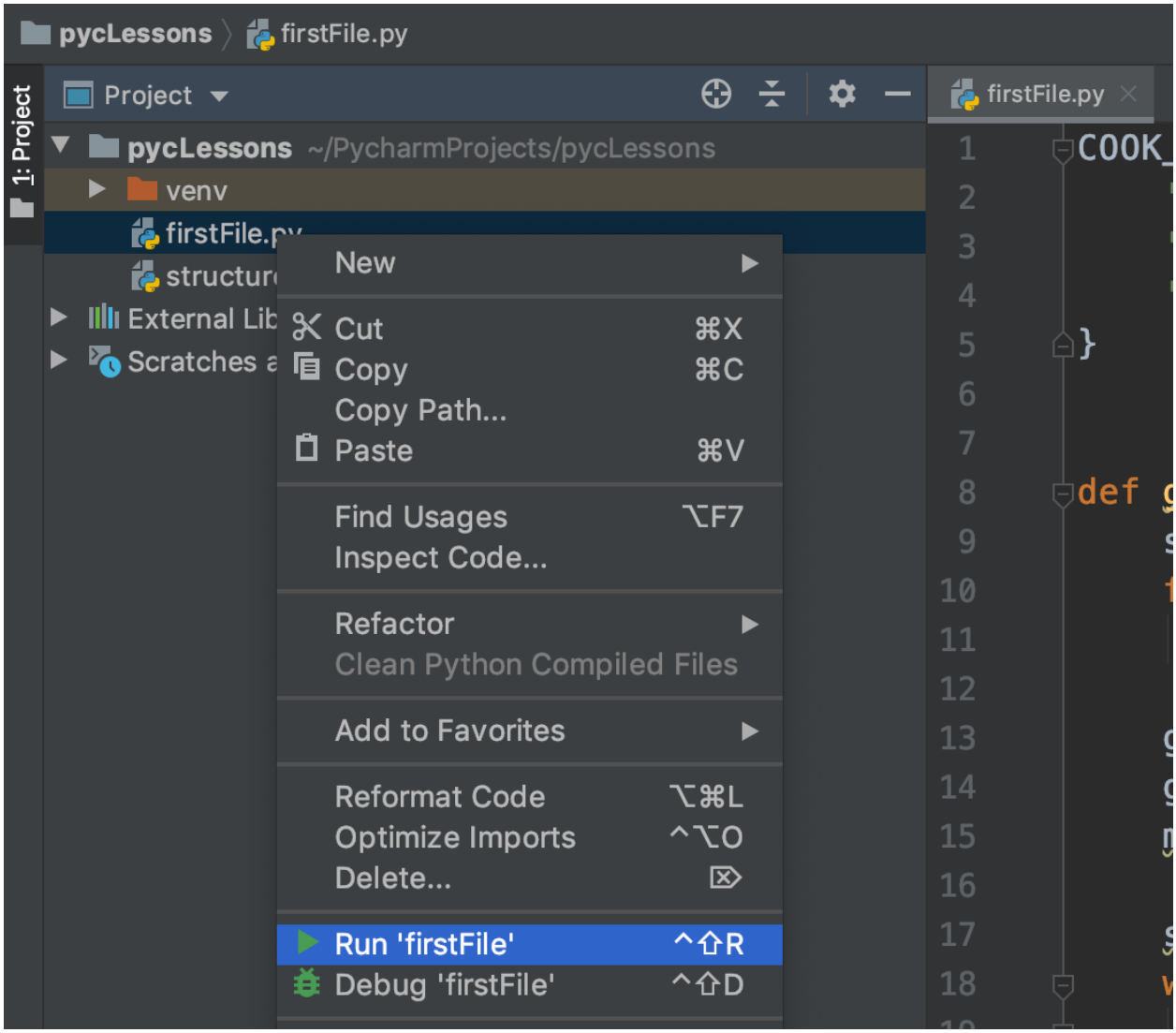
There’s not much to say about this. PyCharm automatically saves files so you don’t have to remember and you’re covered if the program crashes. I’ve added a file to the lesson notes that you can copy into your first file or you can just type print(‘Hello world’) on the first line of the file.

```
COOK_TARGETS = {
    'rare': 120,
    'medium': 130,
    'well done': 140
}


def grillSteak(steakOrders, temperature):
    steaks = []
    for steakOrder in steakOrders:
        steaks.append(Steak(steakOrder))

    grill = Grill(temperature)
    grill.startGrill()
    meatTempIncrease = grill.getTempIncrease()

    steaksNotDone = True
    while steaksNotDone:
        for steak in steaks:
            if not steak.isDone():
                steak.increaseTemperature(meatTempIncrease)
        steaksNotDone = any([not steak.isDone() for steak in steaks])

    print('Steaks are ready to serve')
    return steaks


class Grill:
    def __init__(self, temperatureTarget):
        self.temperature = 0
        self.temperatureTarget = temperatureTarget

    def startGrill(self):
        print('Starting the grill')
        while not self.isReady():
```

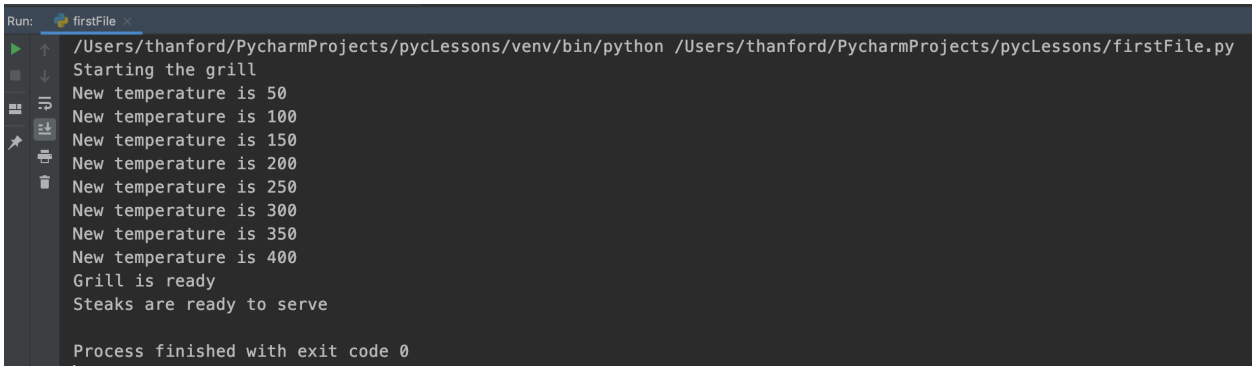
Run a python file

There are a few ways to run a python file in PyCharm. I will cover the two easiest.

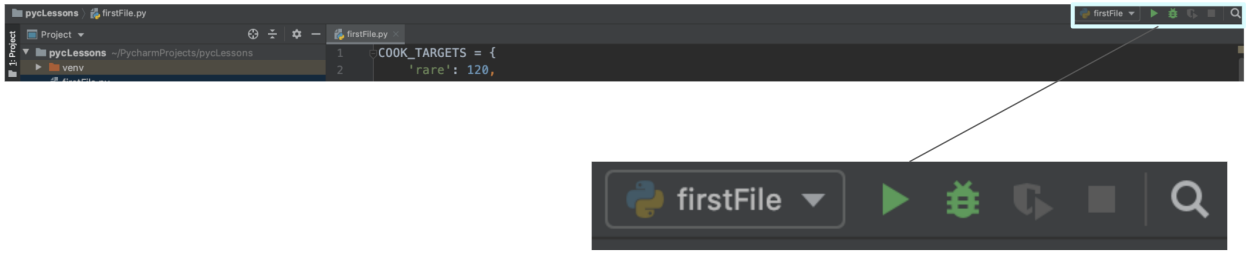
The first option is to right click on the file you want to run in the project bar and select “Run --file name--”. The file name will update based on which file you right clicked on. In my case, it’s “Run firstFile”.



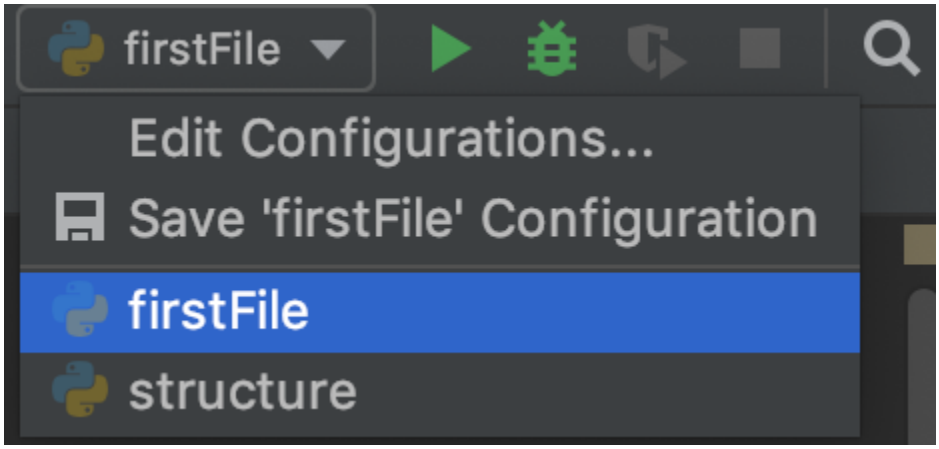
When you run a file for the first time, this is the best option. Once you click “Run”, a “Run” terminal will open at the bottom of the screen. This terminal will print any “print” statements from the file and will also “log” some additional information including which file is being run, any errors that occurred, and the exit status (whether the file ran successfully).



The second way to run a file is to use the file selector and play button at the top right of the PyCharm screen.



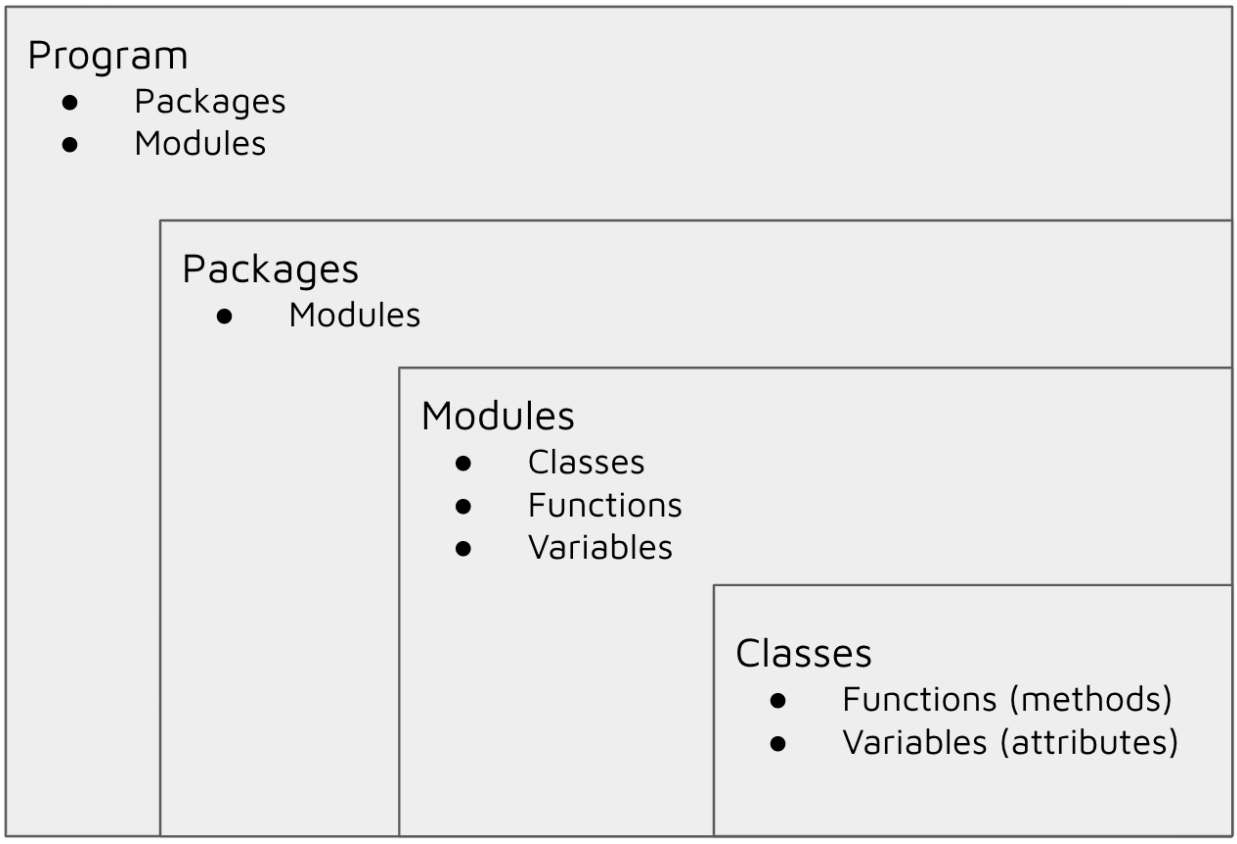
The last file you ran will be shown in the menu, but you can click on the file to open a full list of files that you have previously run.



Select the file you want to run, in this case I'm leaving it as "firstFile", and then click the green play button.

Packages and modules

In the Object Oriented Programming section, we discussed that everything in Python is an object. Well we're about to take that even further! Python files (a.k.a. modules) are objects, and groups of files (a.k.a packages or libraries) are also objects.



Because packages and modules are objects, they can be passed around and used just like classes, functions, or variables would be. We've been making use of modules extensively already! All of the data types that we've used - integers, lists, booleans, etc. - and many of the functions we've used - print, len, etc. - are from the "builtins" module.

Package and module imports

We've already mentioned that packages and modules are objects. In real life, objects have weight and size which limit how many objects can be carried at one time. For example, I am a backpacker and when I go on a multi-day trip, I pay close attention to the weight of my backpack. Excessive weight will drain my energy and slow me down. In Python, objects must be stored in memory so you can think of them as having a "memory weight" associated with them. Because of this, it doesn't make sense to load all objects into your program because they will "weigh down" the program (making it slower). This is where imports come in. Imports allow you to select and use only the objects that you plan to use in the current module.

Packages and modules can come from a few different sources. We can break these down into categories of "always needed", "sometimes needed", and "specialty use".

	Backpacking	Python
Always needed	Food, water, backpack	builtins
Sometimes needed	Knife, hammock, water filter	Standard modules and modules created by user
Specialty use	Crampons, binoculars	Python package index + pip

Always needed
We briefly touched on the "builtins" module. This is a special module that doesn't require you to explicitly import it. Python assumes that these "builtins" are so common and necessary that every module should have access to them.

Sometimes needed
These packages and modules require the use of an "import" statement. They are common enough that they are "pre-installed" when Python is installed, but not common enough that they are loaded into every module. If you are curious which packages and modules are pre-installed, you can Google "Python module index".

Specialty use
These packages and modules have been developed for special purposes and are therefore not "pre-installed" with Python. This requires the use of "pip" to install the package or module and then import it into the necessary module(s). The "Pandas" package is a good example of this. It's very popular, but only used for data analysis. Because not every Python user does data analysis, it is not included with the Python install.

Import statement

As mentioned above, import statements allow us to select the packages and modules that we want to use in the existing module. By convention, import statements are sorted alphabetically at the top of the file.

There are a few different ways to import packages and modules. We can import the entire package or module like this:

```
import collections

myDefaultDict = collections.defaultdict(int)
```

Notice that when the entire package or module is imported, we use dot notation to access the objects within it. In the example above "collections" is a module and "defaultdict" is a class within the module.

If you will be using the same object(s) many times, it may be easier to import specific objects within a package or module.

```
from collections import defaultdict

myDefaultDict = defaultdict(int)
```

Notice that the dot notation is no longer required because defaultdict has been "unpacked" from the collections module.

You can import multiple objects from a module or package by using a comma separated list of objects.

```
from collections import defaultdict, namedtuple

myDefaultDict = defaultdict(int)
myNamedTuple = namedtuple('Lizard', 'scales, tongue, feet')
```

You can also import all objects within a module by using the "*" character, but this is highly discouraged because it can cause very tricky bugs in your code. Some packages or modules may have objects with the same name. If you import both using the "*" character, there will be a "namespace collision" and only one of the objects will win (but it might not be the one you intended to use).

```
from collections import *

myDefaultDict = defaultdict(int)
myNamedTuple = namedtuple('Lizard', 'scales, tongue, feet')
```

You can rename (a.k.a alias) packages, modules, or objects using "as":

```
import collections as col

myDefaultDict = col.defaultdict(int)
myNamedTuple = col.namedtuple('Lizard', 'scales, tongue, feet')
```

```
from collections import defaultdict as dd

myDefaultDict = dd(int)
```

You should avoid renaming most of the time because packages and modules have already been named to convey what they are intended to be used for. Also other individuals reading your code may be familiar with common package and module names, but they certainly won't be familiar with your names.

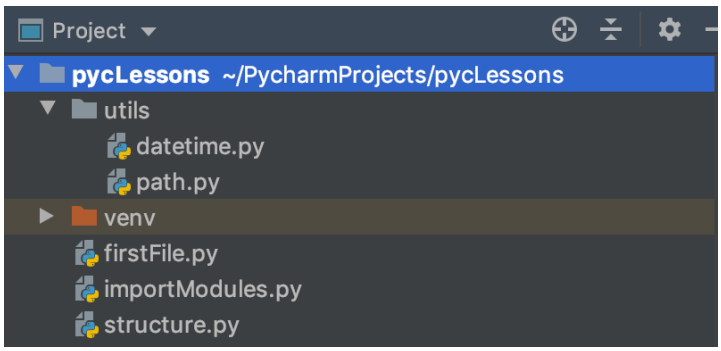
Import file paths

When you import an object, Python only searches for that object name in a few places within your computer's memory. In order, it will search:

1. "builtins" module
2. The current working directory
3. Any directories that have been added to your system's "path"

"Directory" is just another name for a folder. The system's "path" is a list of folder and file paths that will be searched when the system is trying to find something. By having a "path", the system can be efficient when looking for things. Imagine if someone told you to find a gold coin and said that it could be anywhere in the world. That would make your search pretty difficult and you might even give up. Instead, if someone told you the coin was in a drawer in your house, that would be a lot quicker and easier! That's the point of "path". We want to make sure the system knows the spots to look for.

Let's take a few examples. Here is my project structure:



Here is the contents of the path.py file which we will use to understand "paths".

```python
import os
import sys


def printSystemPaths(file):
    currentFilePath = os.path.dirname(os.path.realpath(file))
    print('-------------------------')
    print('Current paths are:')
    for path in set(sys.path):
        if '.egg' not in path and '.zip' not in path:
            if path == currentFilePath:
                print(f'**{path}**')
            else:
                print(path)
    print('-------------------------')
```

I updated the importModules.py file to this:

importModules.py

```python
import collections

from utils import path

path.printSystemPaths(__file__)

print(f'Path for collections is {collections.__file__}')
print(f'Path for path is {path.__file__}')

myDefaultDict = collections.defaultdict(int)
myListLength = len([1, 2, 3])
```

When you run the importModules.py file, you will get the following console logs (not that the current directory has been surrounded with stars "***":

```
/Users/thanford/PycharmProjects/pycLessons/venv/bin/python /Users/thanford/PycharmProjects/pycLessons/importModules.py
------------------------
Current paths are:
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/lib-dynload
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7
/Users/thanford/PycharmProjects/pycLessons/venv/lib/python3.7/site-packages
**/Users/thanford/PycharmProjects/pycLessons**
------------------------
Path for collections is /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/collections/__init__.py
Path for path is /Users/thanford/PycharmProjects/pycLessons/utils/path.py

Process finished with exit code 0
```

Let's start with the "import collections" statement and run down each search location described above. You can see that I printed out the location of collections which is "/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/collections/__init__.py".

1. Is "collections" in the "builtins" module? No
2. Is "collections" in the current directory? No
3. Is "collection" in the system path? Yes

Next, let's look at "from utils import path" which is located at "/Users/thanford/PycharmProjects/pycLessons/utils/path.py".

1. Is "utils" in the "builtins" module? No
2. Is "utils" in the current directory? Yes

Finally, we'll look at the "print" and "len" statements:

1. Are "print" and "len" in the "builtins" module? Yes

You can get more specific with your imports by using dot notation. Remember that you have to start at one of the current paths that the system will search and then go from there. For example:

```
from utils.path import printSystemPaths

printSystemPaths(__file__)
```

First the system will try to find "utils". One of the search locations is "/Users/thanford/PycharmProjects/pycLessons" and "utils" is found within that. Next the system will try to find "path". "path" is a file in the "utils" directory so the system successfully locates it. Finally, the system will search for "printSystemPaths". "printSystemPaths" is a function within the "path" file so the system successfully locates it.

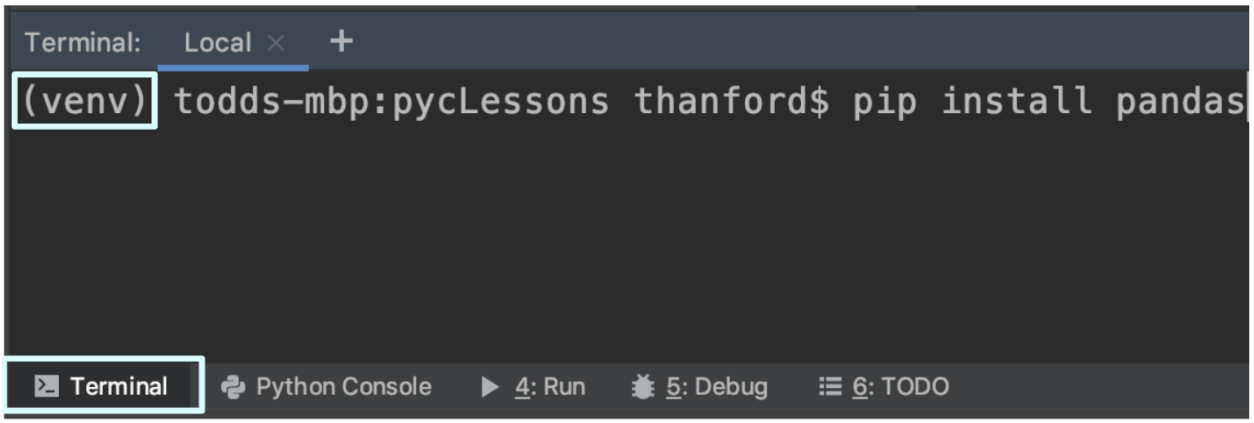
Consider the alternative statement below:

```
from path import printSystemPaths

printSystemPaths(__file__)
```

The system only searches one "layer" deep. It will look in "/Users/thanford/PycharmProjects/pycLessons" and only find "utils", "firstFile", "importModules", and "structure". The system will continue to search in the other locations specified above, but will not find the "path" file.

Install packages

Getting and installing "specialty use" is very simple. If you followed my previous environment set-up guide, you should have a virtual environment installed. Assuming that is the case, you have to make sure the virtual environment is "activated" before you install any packages. I recommend just using the PyCharm terminal because PyCharm is smart enough to activate the virtual environment for you. If you don't use PyCharm, just Google "activate virtual environment" and you will find instructions. For PyCharm users, the "terminal" can be openned with a tab on the bottom left of PyCharm. You will know the virtual environment is activated if the first part of the console line has parentheses with some text in between - "(--virtual environment name--)". In the below example, mine says "(venv)". Yours might say something different. Just make sure there are parentheses.



Once you have openned the terminal and verified that the virtual environment is activated, type "pip install --package name--". You can find an exhaustive list of packages you can download by Googling "Python package index". In the example above, I am installing a package named "pandas". When you hit enter, the terminal will log the steps of the installation and will either indicate that it completed successfully or display an error message.

```
(venv) todds-mbp:pycLessons thanford$ pip install pandas
Collecting pandas
  Downloading https://files.pythonhosted.org/packages/cf/
whl (10.4MB)
    100% |████████████████████████████████| 10.4MB 3.6MB/
Collecting numpy>=1.16.5 (from pandas)
  Downloading https://files.pythonhosted.org/packages/68/
whl (16.0MB)
    100% |████████████████████████████████| 16.0MB 1.7MB/
Collecting pytz>=2017.3 (from pandas)
  Downloading https://files.pythonhosted.org/packages/70/
    100% |████████████████████████████████| 512kB 19.4MB/
Collecting python-dateutil>=2.7.3 (from pandas)
  Using cached https://files.pythonhosted.org/packages/d4
Collecting six>=1.5 (from python-dateutil>=2.7.3->pandas)
  Downloading https://files.pythonhosted.org/packages/ee/
Installing collected packages: numpy, pytz, six, python-d
Successfully installed numpy-1.20.1 pandas-1.2.3 python-d
(venv) todds-mbp:pycLessons thanford$
```

Exception handling

In many cases, you will want your code to stop execution and warn you if an error (a.k.a. exception) occurred. However, there are some good reasons to "handle" certain types of exceptions. In Python exception handling is accomplished the "try/except" statement:

```
try:
    --code block--
except --exception(s)--:
    --code block--
```

For example:

```
>>> keysToTheCastle = {'gold': 'enter', 'brown': 'enter', 'monty': 'enter'}
...
... def breakIntoTheCastle(keyToTry):
...     try:
...         castleKey = keysToTheCastle[keyToTry]
...         print(f'You may {castleKey} the kingdom')
...     except KeyError:
...         print('Thrown in the moat')
...
... breakIntoTheCastle('password')
Thrown in the moat
>>> breakIntoTheCastle('brown')
You may enter the kingdom
>>> keysToTheCastle['password']
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'password'
```

When "breakIntoCastle" is called with "password", the key doesn't exist in "keysToTheCastle". This generates an exception which immediately stops execution of the "try" block. Next, the code checks whether the exception is "excepted". In this case the error is a "KeyError" so the code in the "except" block runs.

When "breakIntoCastle" is called with "brown", the key exists in "keysToTheCastle" so no exception is raised. The "try" block executes to the end. Because there are no exceptions, the "except" block never runs.

Without the "try/except" block, you can see that trying to get a key that doesn't exist in a dictionary will raise a KeyError.

Debugging

Debugging is one of the most important skills in Python or any language. Unfortunately, in my opinion, it is the most under-taught skill. Debugging will give you more control over your code because you won't always have to ask others for help and it will cause you to learn Python better and in more depth than you would otherwise. You can debug with any Python console or editor, but PyCharm has some handy utilities that make debugging much easier. This section will focus on those utilities.

Let's go back to a previous file to use for an example of how to debug code. I'm going to use the "firstFile.py" which was discussed in the "Executing code" section. The code in this file is supposed to take some steak orders, heat up a grill, start grilling steaks when the grill is hot enough, stop grilling steaks when they are at the ordered temperature, and then serve the

steaks. I've made a small modification in my code which causes an error. First I will run the file:

```
/Users/thanford/PycharmProjects/pycLessons/venv/bin/python /Users/thanford/PycharmProjects/pycLessons/firstFile.py
Starting the grill
New temperature is 50
New temperature is 100
New temperature is 150
New temperature is 200
New temperature is 250
New temperature is 300
New temperature is 350
New temperature is 400
Grill is ready
Traceback (most recent call last):
  File "/Users/thanford/PycharmProjects/pycLessons/firstFile.py", line 64, in <module>
    grillSteak(['rare', 'rare', 'well done', 'medium', 'rare', 'medium'], 400)
  File "/Users/thanford/PycharmProjects/pycLessons/firstFile.py", line 20, in grillSteak
    if not steak.isDone():
KeyboardInterrupt

Process finished with exit code 1
```

You can see the file executed with an error. I clicked the red square (stop button) at the top of my screen after the code didn't finish executing for one minute. You can see from the error message that when I clicked the stop button the code was executing line 20 of the file. So that's our first clue...

Debugging with print

I've already added some print statements to the code to indicate the temperature of the grill as it warms up. Print statements can be helpful for you and others to understand what is going on as your code executes, but you don't want to add too many print statements to your code because they are "noisy" and don't alway give you the full picture.

I added a new print statement to the code to see if we can get any more insight.

```
17      steaksNotDone = True
18      while steaksNotDone:
19          for steak in steaks:
20              if not steak.isDone():
21                  print(f'Increasing the steak temperature by {meatTempIncrease}')
22                  steak.increaseTemperature(meatTempIncrease)
23          steaksNotDone = any([not steak.isDone() for steak in steaks])
```

Now when I run the code, the console continuously prints the same thing over and over again.

```
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Traceback (most recent call last):
  File "/Users/thanford/PycharmProjects/pycLessons/firstFile.py", line 65, in <module>
    grillSteak(['rare', 'rare', 'well done', 'medium', 'rare', 'medium'], 400)
  File "/Users/thanford/PycharmProjects/pycLessons/firstFile.py", line 21, in grillSteak
    print(f'Increasing the steak temperature by {meatTempIncrease}')
KeyboardInterrupt
```

So there is clearly something wrong with the "while" loop, but we don't know exactly what yet...

Breakpoints

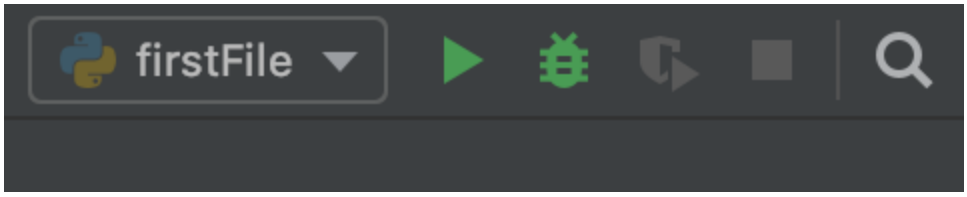
Breakpoints indicate lines in your code where you want to pause code execution. This allows you to understand that state of the entire system at a specific point in time (see how the soup is made). In PyCharm, you can toggle breakpoints by clicking in the left "gutter" of your file (just to the right of the line number):

```
18      while steaksNotDone:
19          for steak in steaks:
20              if not steak.isDone():
21                  print(f'Increasing the steak temperature by {meatTempIncrease}')
22                  steak.increaseTemperature(meatTempIncrease)
23          steaksNotDone = any([not steak.isDone() for steak in steaks])
```

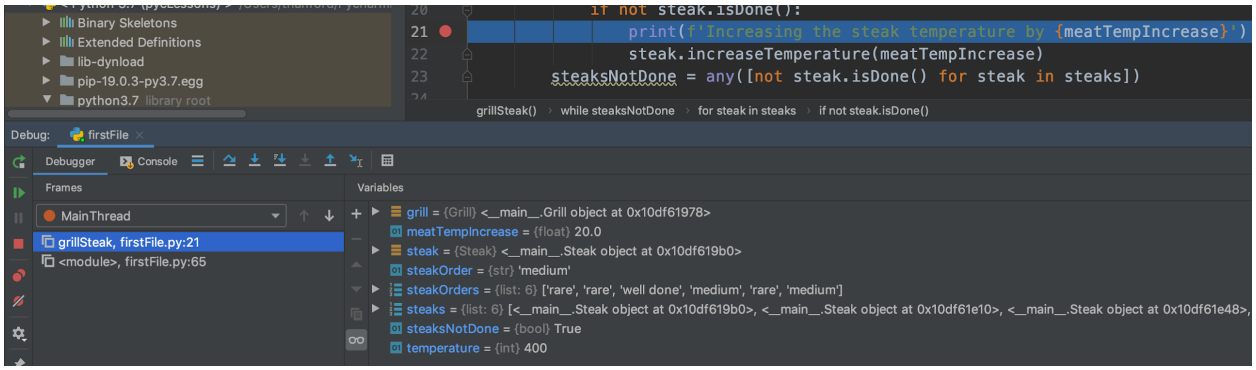
Here, I've clicked next to line 21 to set a breakpoint there.

Running in "debug" mode

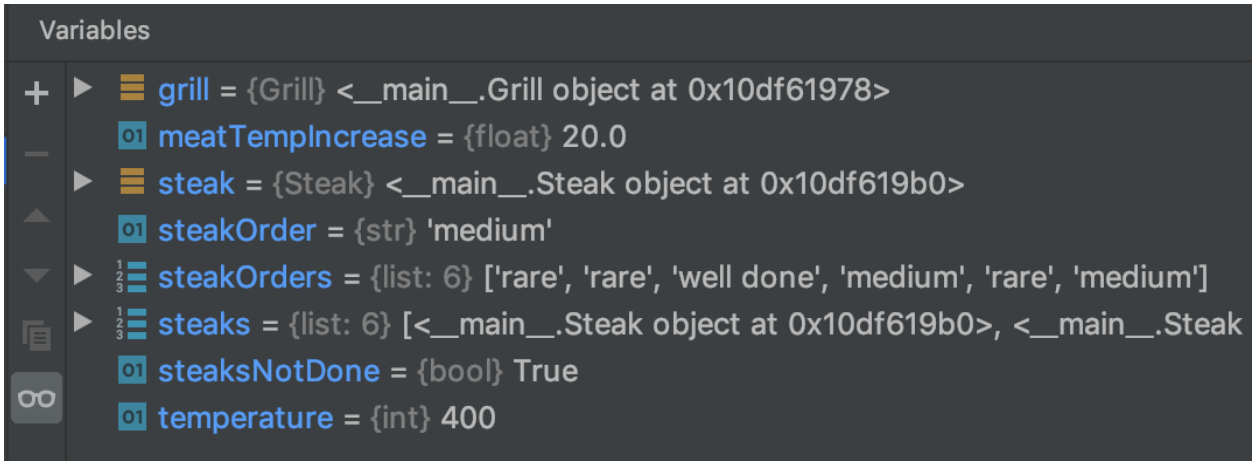
Breakpoints only work when you run your code in "debug" mode. This is accomplished by clicking the green bug icon at the top right of the PyCharm screen:



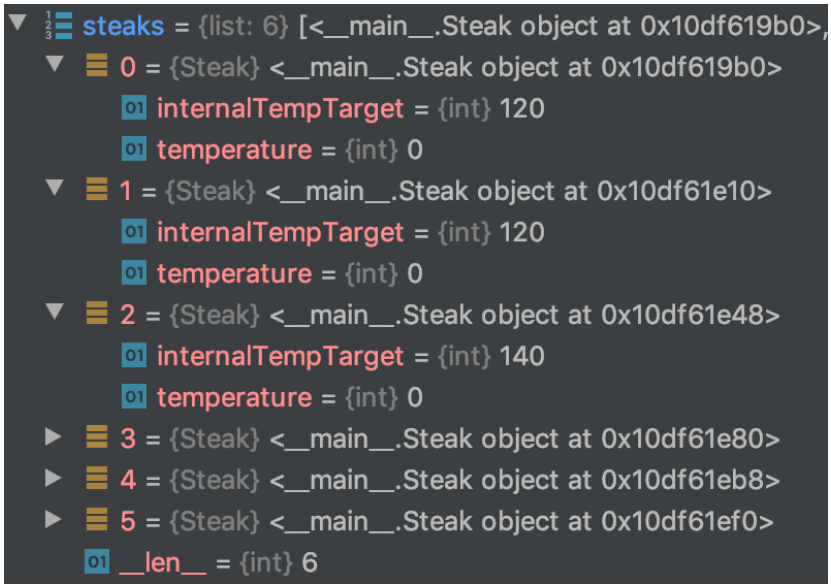
When I run the file in debug mode, code execution stops at the breakpoint I set on line 21:



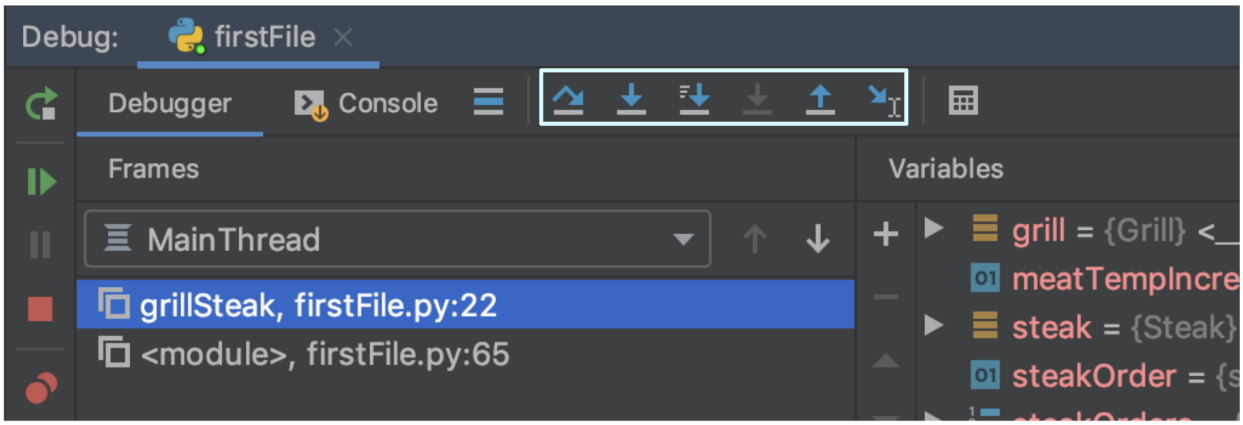
You'll notice that line 21 is highlighted and a "debugger" panel is displayed at the bottom of the screen. The debugger panel shows all of the variables and their values at the current point in execution (line 21).



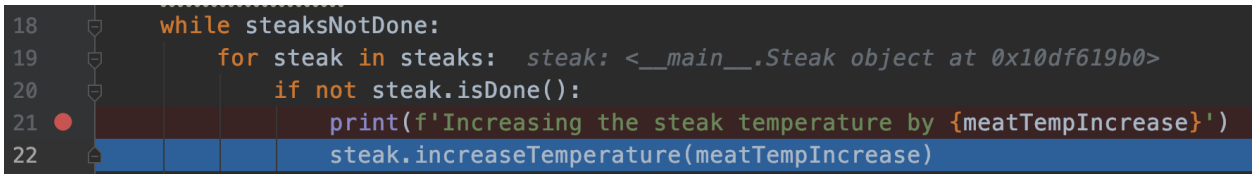
Any variables with an arrow next to them indicate that you can inspect them further. When I inspect the "steaks" variable, I can see that every steak has a temperature of 0.



There are a few arrows at the top of the debugger panel that allow me to control how code executes from the line where the breakpoint was hit. You can hover over each arrow which will show a text box of what the arrow does.



I clicked the first arrow which is used to "step over". This tells PyCharm to go to the next line in execution (line 22).

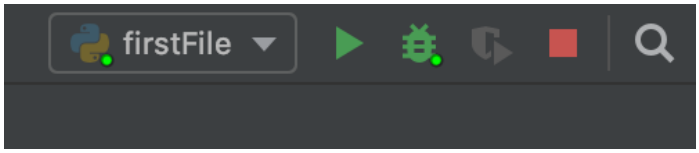


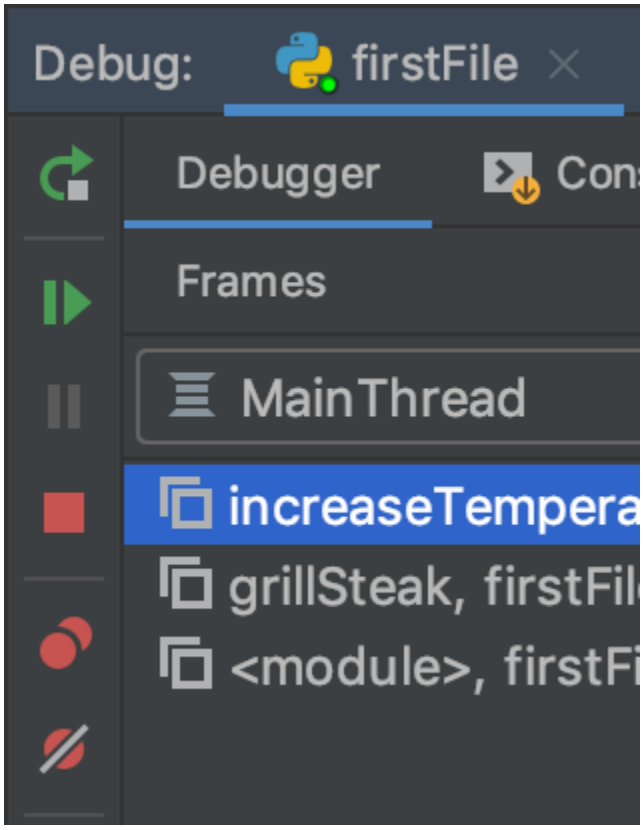
I want to see what happens when the "increaseTemperature" method is run so I clicked on the second arrow to "step into" the method on the current line (line 22).

```
53  class Steak:
54      def __init__(self, cookTarget):
55          self.temperature = 0
56          self.internalTempTarget = COOK_TARGETS[cookTarget]
57
58      def increaseTemperature(self, tempIncrease):  self: <__
59          self.temperature = tempIncrease
```

This takes me to the first line where "increaseTemperature" is executed. From here, I can see the error which is occurring - the "temperature" attribute is being assigned to "tempIncrease", but it really should be adding "tempIncrease" to the current "temperature".

Now that I understand the issue, I can stop the code from running any further by clicking the red square at the top right of the PyCharm screen or the red square in the left side of the debugger panel.





Now I update my code to fix the error:

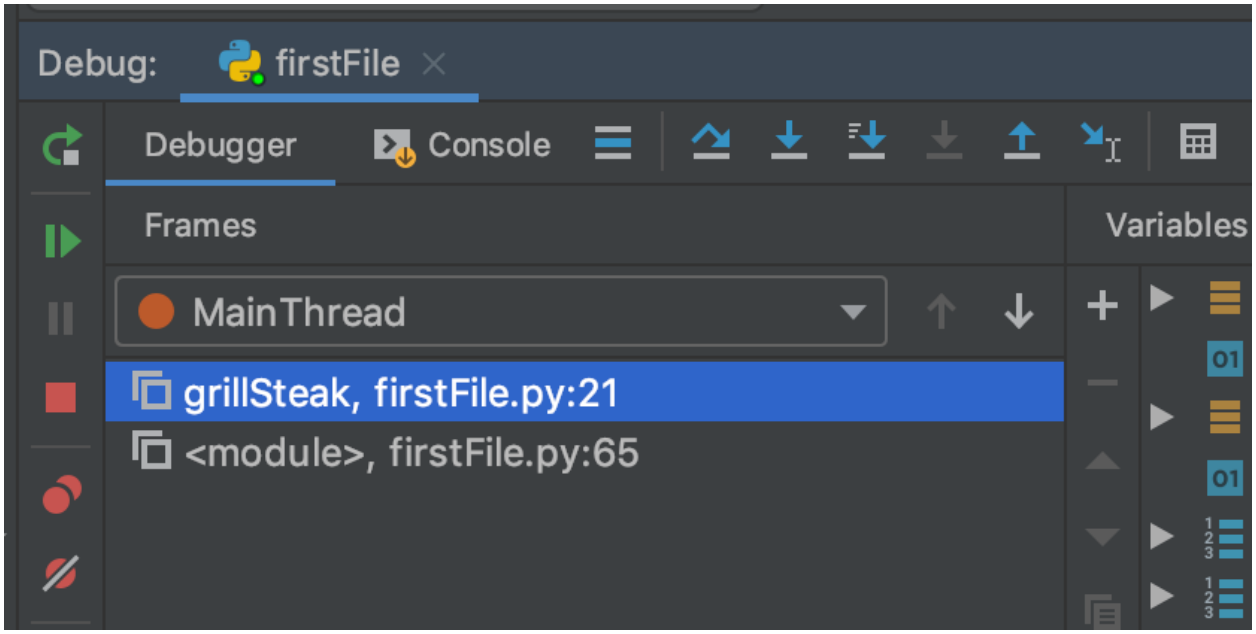
```
58      def increaseTemperature(self, tempIncrease):
59          self.temperature += tempIncrease
```

And then I run the file in normal mode (by clicking the green triangle). From there I can see that the code runs successfully:

```
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Increasing the steak temperature by 20.0
Steaks are ready to serve

Process finished with exit code 0
```

Handy debug tools

There are a few debug tools worth mentioning in the PyCharm debugger control panel:



 "Step over" runs the next line of code in the current file you are in.

 "Step into" and "Step into my code" both enter the execution of a function or method on the current line. "Step into my code" will ignore installed packages.

 "Step out" will move up from a function or method that is being executed to the line of code where that function or method was called.

 "Run to cursor" will execute code up to the point where the cursor is.

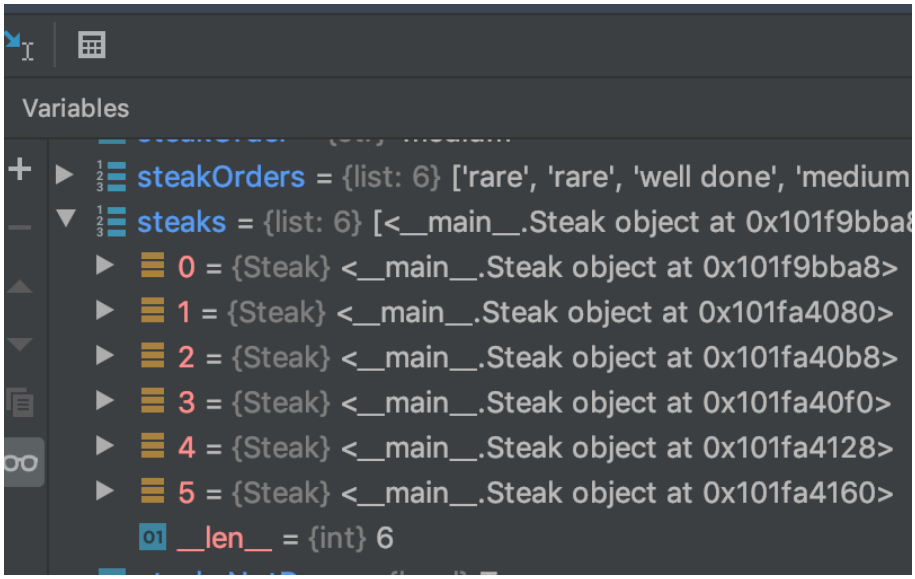
 "Resume program" will cause code to execute up to the next breakpoint or until the program finishes; whichever comes first.

 "Mute breakpoints" will toggle whether breakpoints will be hit. When you toggle for the first time, you will see your breakpoint(s) turn color from red to grey (indicating that they are no longer active).

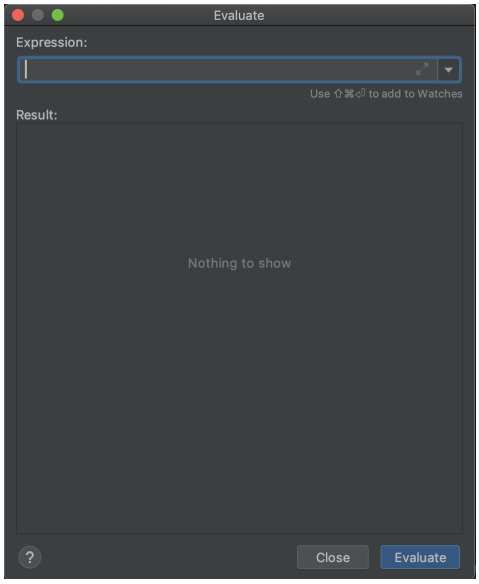
If you click the button again, breakpoint(s) will toggle back to the red color (indicating they are active).

If breakpoints are inactive and you click the "Resume program" button, the code will execute without hitting any more breakpoints.

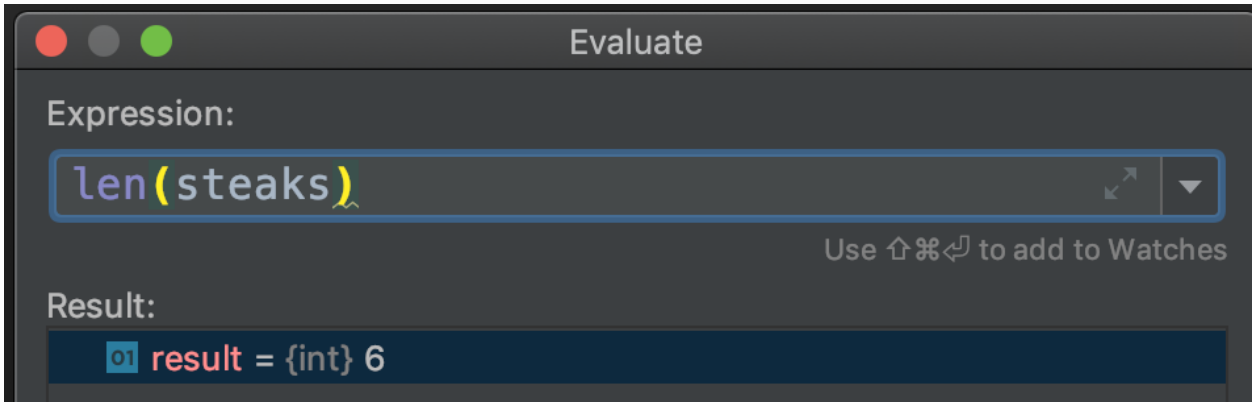
 "Evaluate" lets you interact with all of the variables in the current state (at the current line that has been executed). For example, At the current breakpoint, steaks is a list of six instances of "Steak":



If I click the "Evaluate" icon, a dialog box opens up:



In the box at the top, I can type in any code that can make use of any variables like "steaks":



Comments

In the variables section we talked about naming variables to indicate meaning. Even if you come up with the best names for your variables, if you step away from your code for a week or two and come back to it, there's a good chance it will take you some time to figure out exactly what your code is doing. Imagine if someone who hadn't written the code tried to jump in to figure out what you were doing! This is why code comments are so important. They are like little memos or bread crumbs that help you (and others) quickly understand what your code is doing. Adding comments to your code is very easy to ignore because it takes extra work and doesn't have immediate benefit (because you hopefully should know what your code is doing when you write it), but you will thank yourself later.

In-line comments

In-line comments are short comments related to the code on the same line or immediately underneath it. These comments should indicate something non-obvious. Adding comments is an art, not a science, so you just have to use your judgment to determine when a comment is necessary. Overall, it is MUCH better to over-comment than under-comment.

Single line comments are added by using the hash sign (#). Everything after the hash sign will be considered part of the comment.

```
steaks = []
for steakOrder in steakOrders:
    steaks.append(Steak(steakOrder))  # Create a Steak object based on each order
```

You can also add multiple lines of comments.

```
# Increase the temperature of each steak
# Once all steaks are done, exit the loop
steaksNotDone = True
while steaksNotDone:
    for steak in steaks:
```

If you are writing more than a couple lines, you can surround your comment with triple quotes.

```
"""Increase the temperature of each steak.
Once all steaks are done, exit the loop.
I really like steaks. And I could go on for many
lines about the right way to cook steaks.
"""
steaksNotDone = True
while steaksNotDone:
```

[Advanced] Docstrings

Docstrings allow you to comment on a module, class, function, or method. The intention of a docstring is to indicate what the code in the object is intended to do and to provide more detail on the inputs and outputs.

Docstrings are created using triple quotes. When you add triple quotes just below an object, PyCharm will auto-complete with the starting structure of a docstring.

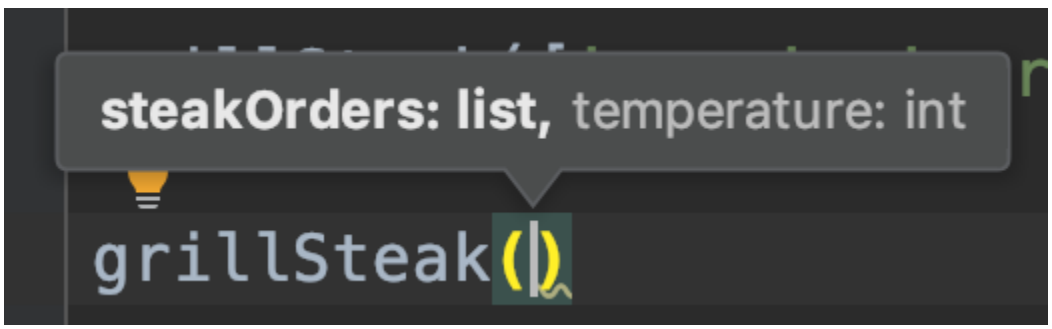
The first part of the docstring should be a high level summary of what the object is intended to do. The next part of the docstring should indicate any parameters using the following syntax: "`:param --data type of parameter-- --name of parameter--: --description of parameter--`". Finally, the last part of the docstring should indicate the return value using the following syntax: "`:return --data type of return value--: --description of return value--`".

```
def grillSteak(steakOrders, temperature):
    """Get a list of Steaks, each heated to the ordered cook target.

    :param list steakOrders: A list of steak orders. Each item must be a key in COOK_TARGETS
    :param int temperature: An integer indicating the tem
    :return list: A list of Steak objects heated to the temperature based on the order.
    """
    steaks = []
    for steakOrder in steakOrders:
        steaks.append(Steak(steakOrder))  # Create a Steak object based on each order
```

All good Python packages add docstrings. For example, here is the docstring for the builtin print function:

```
def print(self, *args, sep=' ', end='\n', file=None): # known special case of 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.
    """
```

Docstrings are an amazing way to make sure code is understandable. Additionally, PyCharm will provide helpful context when you or anyone else uses an object that has a docstring. When I type in the start of the function I just documented, I get the parameters and data types:



PyCharm allows you to see the full docstring in a popover window if you use the "option + spacebar" keys: