	Function: Description: int(x) Converts x to an integer whole number float(x) Converts x to a floating-point number str(x) Converts x to a string representation chr(x) Converts integer x to a character unichr(x) Converts integer x to a Unicode character ord(x) Converts character x to its integer value hex(x) Converts integer x to a hexadecimal string oct(x) Converts integer x to an octal string
	You can check what type of object is assigned to a variable using Python's built-in type() function. Common data types include: ● **int** (for integer) ● **float** ● **str** (for string) ● **list** ● **tuple** ● **dict** (for dictionary) ● **set** ● **bool** (for Boolean True/False)
Types of data container in Python:	**Variable – stores a single value.** **List – stores multiple values in an ordered index.** **Tuple – stores multiple fixed values in a sequence.** **Set –stores multiple unique values in an unordered collection.** **Dictionary – stores multiple unordered key:value pairs.**

	Looping over items
	`for num in list1:`
	`    print(num)`

```
for letter in 'This is a.':
    print(letter)
```

Functions

Functions

```
def name_of_function(arg1,arg2=defaultvalue):
    '''
    This is where the function's Document String (docstring) goes
    '''
    # Do stuff here
    # Return desired result
```

*args

```
def myfunc(*args):
    return sum(args)
```

When a function parameter starts with an asterisk, it allows for an *arbitrary number* of arguments, and the function takes them in as a tuple of values.

**kwargs

Python offers a way to handle arbitrary numbers of keyworded arguments. Instead of creating a tuple of values, **kwargs builds a dictionary of key/value pairs also known as keyworded arguments.

```
def myfunc(**kwargs):
    if 'fruit' in kwargs:
        print(f"My favorite fruit is {kwargs['fruit']}")
    else:
        print("I don't like fruit")

myfunc(fruit='pineapple')
```

map function

The **map** function allows you to "map" a function to an iterable object. That is to say you can quickly call the same function to every item in an iterable, such as a list. For example:

```
def square(num):
    return num**2

my_nums = [1,2,3,4,5]

map(square,my_nums)
```

Error: <map at 0x205baec21d0>

```
list(map(square,my_nums))
```

[1, 4, 9, 16, 25]

filter function

The filter function returns an iterator yielding those items of iterable for which function(item) is true. Meaning you need to filter by a function that returns either True or False. Then passing that into filter (along with your iterable) and you will get back only the results that would return True when passed to the function.

Pass an iterable item (list) along with a function which return a true or false to filter function, a list of items for which function returned a true value will be created.

```
def check_even(num):
    return num % 2 == 0

nums = [0,1,2,3,4,5,6,7,8,9,10]

list(filter(check_even,nums))
```

[0, 2, 4, 6, 8, 10]

lambda expression

```
def square(num):
    return num**2
```

Or

```
def square(num): return num**2

lambda num: num ** 2
```

A lambda function that multiplies argument a with argument b and print the result:

```
x = lambda a, b : a * b
```

	print(x(5, 6))
	30
	list(map(lambda num: num ** 2, my_nums))
	[1, 4, 9, 16, 25]

Importing modules	
Managing strings	
+	Join strings
*	Repeat strings letter = 'z' letter*10 'zzzzzzzzzz'
[:] [start:stop:stepsize]	Select char in specified range mychar=”01234567” mychar[:3] will be 012 (will stop at 3) Mychar[3:6] will be 345 Mychar[::] will print whole string Mychar [::2] step size 2 0246 Mychar[::-1] will reverse the string
in	Return true if character ‘H’ exists in ‘Hello’
‘‘‘ ‘‘‘	Describe a module function class or method E.g. def display(s) : ‘’’Display an argument value.’’’ print(s)
Split Format method	

Change case	1.capitalize( )
	2.title( )
	3.upper( )
	4.lower( )

Remove white space	1. lstrip( ) 2. rstrip ( ) 3.strip()
Find and replace	1. replace(old , new) Replace all occurance of old with new 2. count(sub) Return the number of occurrences of sub, or return -1 3. find (sub) Return the index number of first occurence of sub or -1

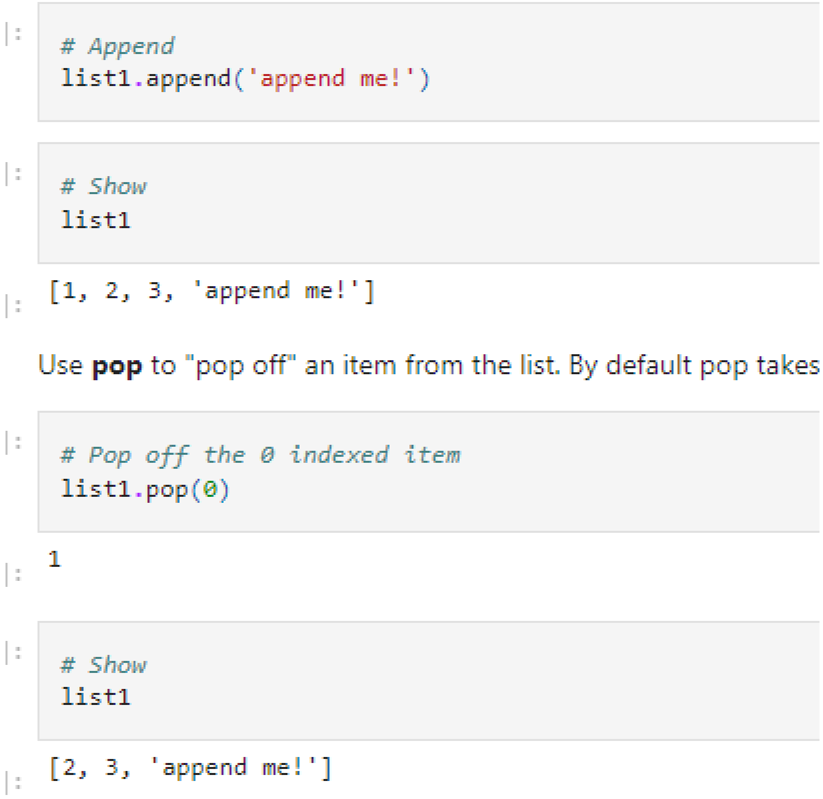
Is ?	1. isalpha()
	2. isnumeric()
	3.isalnum()
	4.islower()
	5.isuper()
	6.istitle()
	7. isdigit()
	8. isdecimal()

Lists:

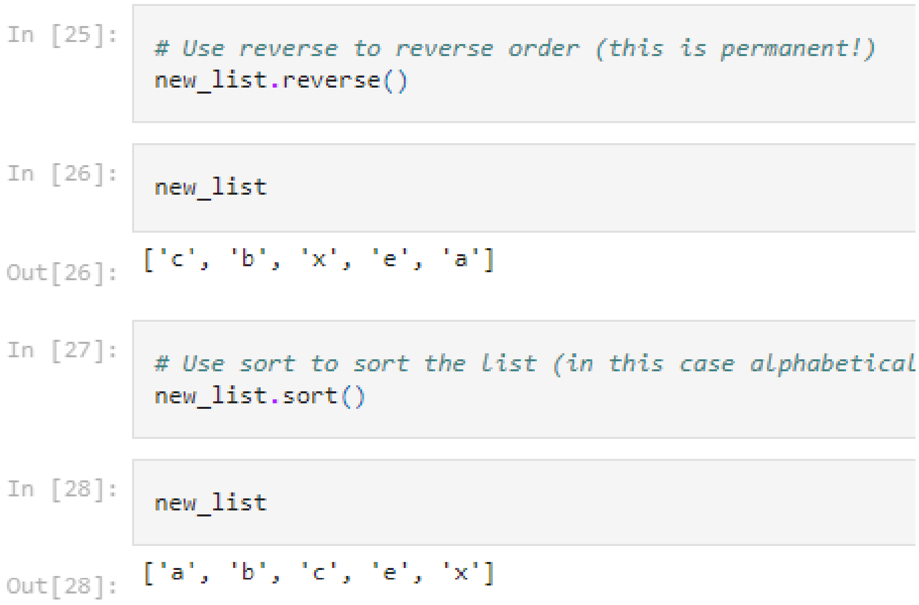
Creating lists	nums = [0 , 1 , 2 , 3 , 4 , 5]
Indexing and Slicing	my_list = ['one','two','three',4,5] # Grab index 1 and everything past it my_list[1:] ['two', 'three', 4, 5] # Grab everything UP TO index 3 my_list[:3] ['one', 'two', 'three']

Check if a list contains an element	The in operator will return True if a specific element is in a list.
	`li = [1,2,3,'a','b','c']`
	`'a' in li`
	`#=> True`

Add elements to list:	

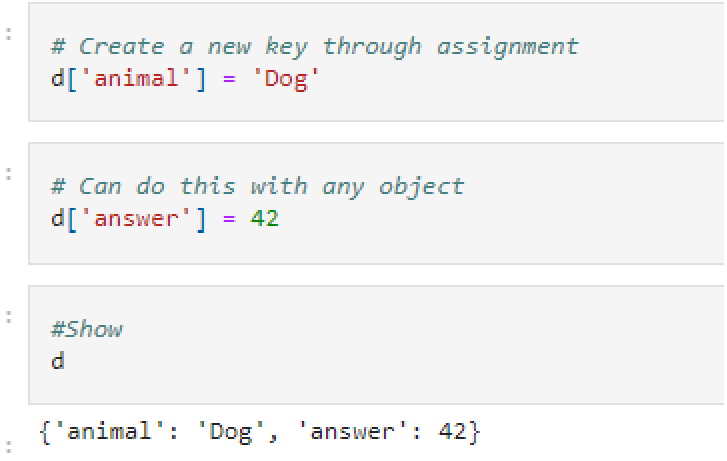
	1. list.append(x)
	2. list.extend(L) Adds all items in list L to the end of the list
	3. list.insert(i,x) Inserts item x at index position i
	
	`# Append`
	`list1.append('append me!')`
	`# Show`
	`list1`
	`[1, 2, 3, 'append me!']`
	Use **pop** to "pop off" an item from the list. By default pop takes
	`# Pop off the 0 indexed item`
	`list1.pop(0)`
	`1`
	`# Show`
	`list1`
	`[2, 3, 'append me!']`

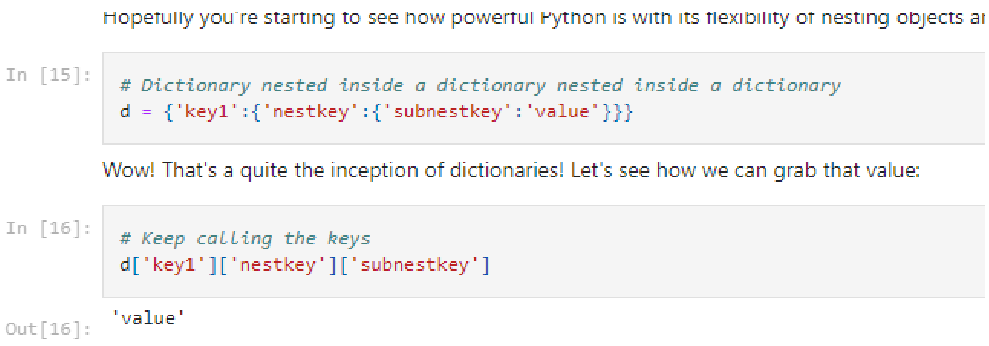
Remove elements	1. list.remove(x) Removes first item x from the list 2. list.pop(i) Removes item at index position i and returns it
Search:	1. list.index(x) Returns the index position in the list of first item x 2. list.count(x) Returns the number of times x appears in the list
Sort:	1. list.sort() Sort all list items, in place 2. list.reverse() Reverse all list items, in place

	
	In [25]: `# Use reverse to reverse order (this is permanent!)`
	`new_list.reverse()`
	In [26]: `new_list`
	Out[26]: `['c', 'b', 'x', 'e', 'a']`
	In [27]: `# Use sort to sort the list (in this case alphabetical`
	`new_list.sort()`
	In [28]: `new_list`
	Out[28]: `['a', 'b', 'c', 'e', 'x']`

Dictionaries:

Constructing a Dictionary	*# Make a dictionary with {} and : to signify a key and a value* my_dict **=** {'key1':'value1','key2':'value2'} *# Call values by their key* `my_dict['key2']` value2 dictionaries are very flexible in the data types they can hold. my_dict **=** {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}

	my_dict['key3'] ['item0', 'item1', 'item2'] my_dict['key3'][0] 'Item0' We can also create keys by assignment.  # Create a new key through assignment d['animal'] = 'Dog' # Can do this with any object d['answer'] = 42 #Show d {'animal': 'Dog', 'answer': 42}

Nesting with Dictionaries	
	Hopefully you're starting to see how powerful Python is with its flexibility of nesting objects ar
	In [15]: # Dictionary nested inside a dictionary nested inside a dictionary
	d = {'key1':{'nestkey':{'subnestkey':'value'}}}
	Wow! That's a quite the inception of dictionaries! Let's see how we can grab that value:
	In [16]: # Keep calling the keys
	d['key1']['nestkey']['subnestkey']
	Out[16]: 'value'

Dictionary Methods	*# Method to return a list of all keys*
	d.keys()
	# Method to grab all values
	d.values()
	# Method to return tuples of all items (we'll learn about tuples soon)
	d.items()

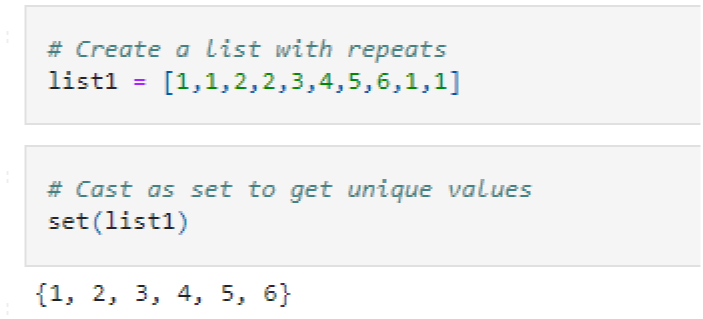
Tuple:

Creating Tuple	t = (1,2,3) t[0] 'one' *# Can also mix object types* t **=** ('one',2)
Immutability	Because of this immutability, tuples can't grow. Once a tuple is made we can not add to it. t.append('nope')
Basic Tuple Methods	*# Use .index to enter a value and return the index* t**.**index('one') *# Use .count to count the number of times a value appears* t**.**count('one')

Sets:

Creating sets	S = {1, 2, 3, 4, 5, 6}
	list1 = [1,1,2,2,3,4,5,6,1,1]
	set(list1)

Set functions:	set.add(x) Adds item x to the set *# We add to sets with the add() method* x**.**add(1)

<table>
<tr><td></td><td>


```
# Create a list with repeats
list1 = [1,1,2,2,3,4,5,6,1,1]
```

```
# Cast as set to get unique values
set(list1)
```

```
{1, 2, 3, 4, 5, 6}
```

set.update(x,y,z) Adds multiple items to the set
set.copy() Returns a copy of the set
set.pop() Removes one random item from the set
set.discard(x) Removes item x if found in the set
set1.intersection(set2) Returns items that appear in both sets
set1.difference(set2) Returns items in set1 but not in set2
</td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td></tr>
</table>

Common methods across List, Tuple and Sets:

len()
max()
min()
reversed()
sorted()
sum()	

any() (if any element is true)	

all() (if all elements are true) bool() (convert to bool) filter() constructs iterator from elements which are true	
enumerate()	
filter()	
iter()	
map()	
slice()	
zip()	

Accessing files

Opening Files in Python	`>>> f = open("test.txt")    # open file in current directory`
	`>>> f = open("C:/Python38/README.txt")  # specifying full`
	`path`
	
	Mode
	R
	W
	X
	A
	T (text mode)
	B (binary mode)
	+ Opens a file for updating (reading and writing)
	
	`with open("test.txt", encoding = 'utf-8') as f:`
	`   # perform file operations`

Closing Files in Python	`f = open("test.txt", encoding = 'utf-8')`
	`# perform file operations`
	`f.close()`
	
	`try:`
	`   f = open("test.txt", encoding = 'utf-8')`
	`   # perform file operations`

<table>
<tr>
<td></td>
<td>

```
finally:
   f.close()
```

The best way to close a file is by using the with statement. This ensures that the file is closed when the block inside the with statement is exited.
We don't need to explicitly call the close() method. It is done internally.

```
with open("test.txt", encoding = 'utf-8') as f:
   # perform file operations
```

</td>
</tr>
<tr>
<td>Writing to Files in Python</td>
<td>

In order to write into a file in Python, we need to open it in write w, append a or exclusive creation x mode.
We need to be careful with the w mode, as it will overwrite into the file if it already exists. Due to this, all the previous data are erased.

```
with open("test.txt",'w',encoding = 'utf-8') as f:
   f.write("my first file\n")
   f.write("This file\n\n")
   f.write("contains three lines\n")
```

</td>
</tr>
<tr>
<td>Reading Files in Python</td>
<td>

There are various methods available for this purpose. We can use the read(size) method to read in the size number of data. If the size parameter is not specified, it reads and returns up to the end of the file.

```
>>> f = open("test.txt",'r',encoding = 'utf-8')
>>> f.read(4)    # read the first 4 data
'This'

>>> f.read(4)    # read the next 4 data
' is '

>>> f.read()     # read in the rest till end of file
'my first file\nThis file\ncontains three lines\n'

>>> f.read()  # further reading returns empty sting
''
```

We can change our current file cursor (position) using the `seek()` method. Similarly, the `tell()` method returns our current position (in number of bytes).

```
>>> f.tell()    # get the current file position
56

>>> f.seek(0)   # bring file cursor to initial position
0
```

</td>
</tr>
</table>

```
>>> print(f.read())   # read the entire file
This is my first file
This file
contains three lines
```

We can read a file line-by-line using a for loop. This is both efficient and fast.

```
>>> for line in f:
...     print(line, end = '')
...
This is my first file
This file
contains three lines
```

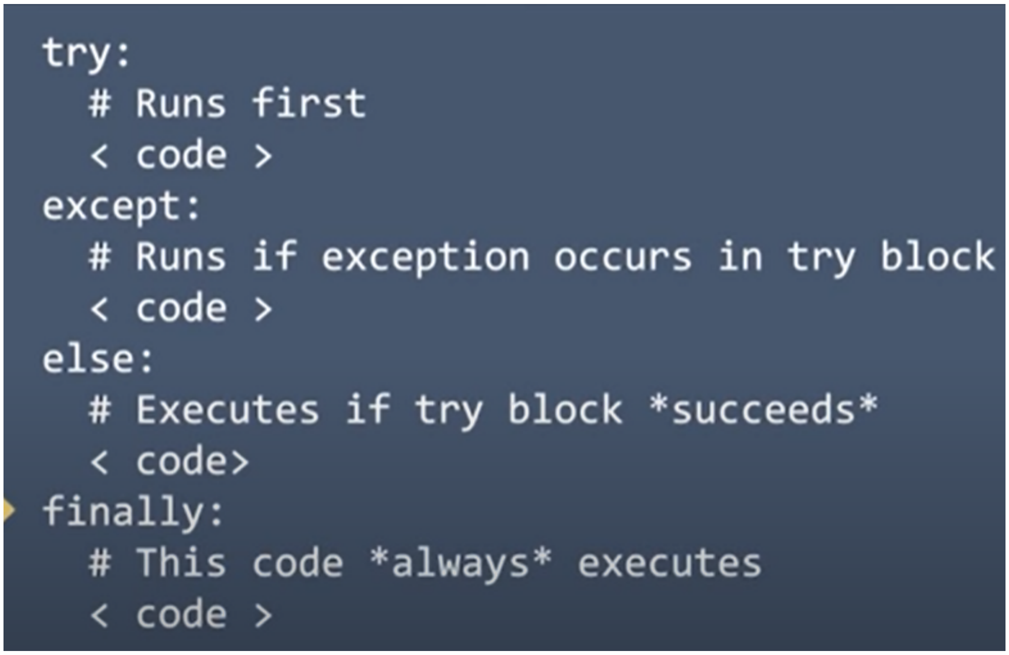
Alternatively, we can use the `readline()` method to read individual lines of a file. This method reads a file till the newline, including the newline character.

```
>>> f.readline()
'This is my first file\n'

>>> f.readline()
'This file\n'

>>> f.readline()
'contains three lines\n'

>>> f.readline()
''
```

Lastly, the `readlines()` method returns a list of remaining lines of the entire file. All these reading methods return empty values when the end of file (EOF) is reached.

```
>>> f.readlines()
['This is my first file\n', 'This file\n', 'contains three
lines\n']
```


Exeception

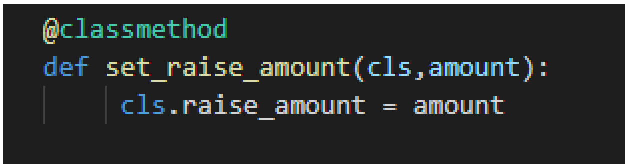
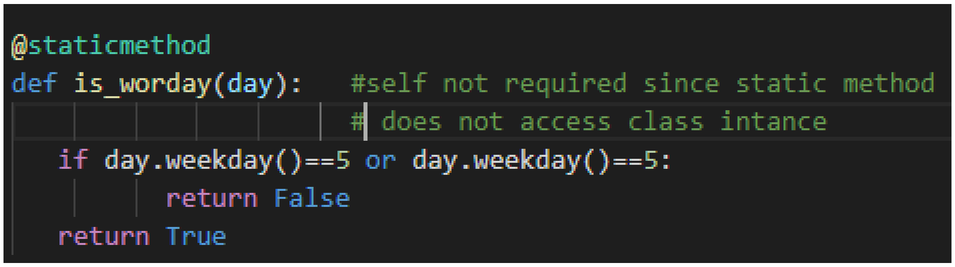
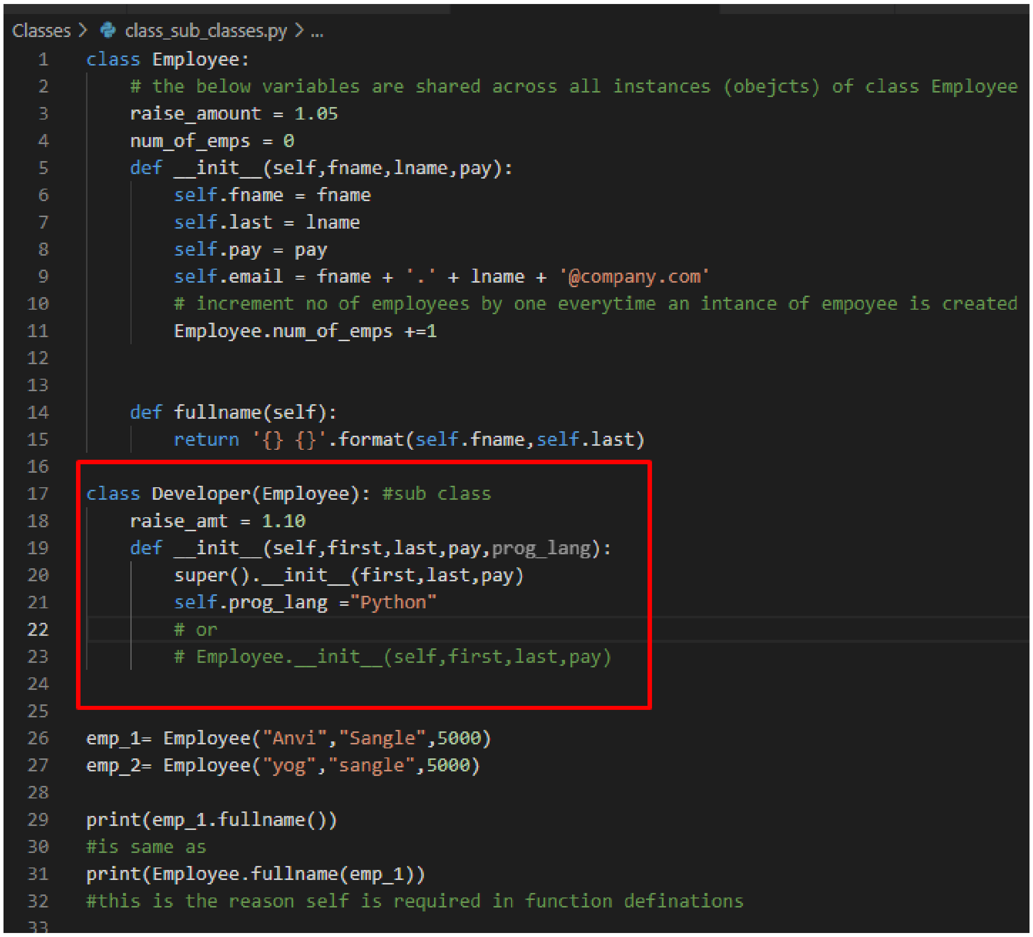
	

```
try:
  # Runs first
  < code >
except:
  # Runs if exception occurs in try block
  < code >
else:
  # Executes if try block *succeeds*
  < code>
finally:
  # This code *always* executes
  < code >
```

Object Oriented Programming

Class	User defined objects are created using the class keyword. The class is a blueprint that defines the nature of a future object. From classes we can construct instances. An instance is a specific object created from a particular class.
	
	# Create a new object type called Sample
	class Sample:
	    pass
	
	# Instance of Sample
	x = Sample()
	
	print(type(x))
	
	<class '__main__.Sample'>
	
	An **attribute** is a characteristic of an object. A **method** is an operation we can perform with the object.

Attributes	There is a special method called:
	
	__init__()
	
	This method is used to initialize the attributes of an object
	
	The syntax for creating an attribute is:
	
	self.attribute = something
	
	class Dog:
	    def __init__(self,breed):
	        self.breed = breed
	
	The special method __init__() is called automatically right after the object has been created.
	
	Whenever an object of class is created __init__ method under the class is called with class object as as a parameter. That is why you need self as an argument in __init__ method.

<table>
<tr>
<td>Class methods</td>
<td>

<pre><code>@classmethod
def set_raise_amount(cls,amount):
    cls.raise_amount = amount</code></pre>
</td>
</tr>
<tr>
<td>Static Methods:</td>
<td>

<pre><code>@staticmethod
def is_worday(day):   #self not required since static method
                       # does not access class intance
   if day.weekday()==5 or day.weekday()==5:
        return False
   return True</code></pre>
Static method does not access instance variables.
</td>
</tr>
<tr>
<td>Sub classes</td>
<td>


<pre><code>class Employee:
    # the below variables are shared across all instances (obejcts) of class Employee
    raise_amount = 1.05
    num_of_emps = 0
    def __init__(self,fname,lname,pay):
        self.fname = fname
        self.last = lname
        self.pay = pay
        self.email = fname + '.' + lname + '@company.com'
        # increment no of employees by one everytime an intance of empoyee is created
        Employee.num_of_emps +=1

    def fullname(self):
        return '{} {}'.format(self.fname,self.last)

class Developer(Employee): #sub class
    raise_amt = 1.10
    def __init__(self,first,last,pay,prog_lang):
        super().__init__(first,last,pay)
        self.prog_lang ="Python"
        # or
        # Employee.__init__(self,first,last,pay)

emp_1= Employee("Anvi","Sangle",5000)
emp_2= Employee("yog","sangle",5000)

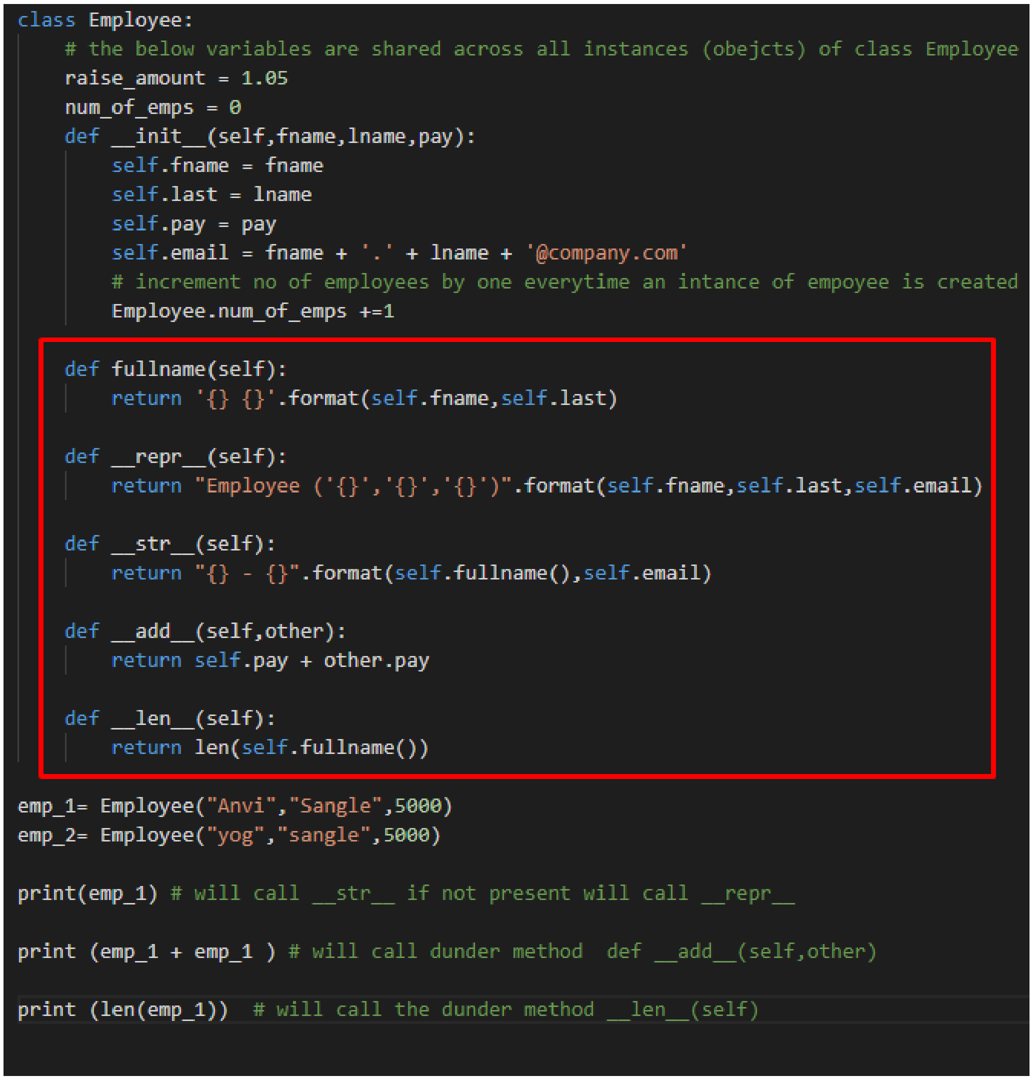
print(emp_1.fullname())
#is same as
print(Employee.fullname(emp_1))
#this is the reason self is required in function definations</code></pre>
</td>
</tr>
</table>

Special methods
These methods change how objects are printed.
def __repr__(self):

def __str__(self):

def __add__

def __len__



```
class Employee:
    # the below variables are shared across all instances (obejcts) of class Employee
    raise_amount = 1.05
    num_of_emps = 0
    def __init__(self,fname,lname,pay):
        self.fname = fname
        self.last = lname
        self.pay = pay
        self.email = fname + '.' + lname + '@company.com'
        # increment no of employees by one everytime an intance of empoyee is created
        Employee.num_of_emps +=1

    def fullname(self):
        return '{} {}'.format(self.fname,self.last)

    def __repr__(self):
        return "Employee ('{}','{}','{}')".format(self.fname,self.last,self.email)

    def __str__(self):
        return "{} - {}".format(self.fullname(),self.email)

    def __add__(self,other):
        return self.pay + other.pay

    def __len__(self):
        return len(self.fullname())

emp_1= Employee("Anvi","Sangle",5000)
emp_2= Employee("yog","sangle",5000)

print(emp_1) # will call __str__ if not present will call __repr__

print (emp_1 + emp_1 ) # will call dunder method  def __add__(self,other)

print (len(emp_1))  # will call the dunder method __len__(self)
```

Property Decorators - Getters, Setters, and Deleters

https://www.youtube.com/watch?v=jCzT9XFZ5bw&list=PL-osiE80TeTsqhIuOqKhwlXsIBIdSeYtc&index=6

String Formatting

There are three ways to perform string formatting.

- The oldest method involves placeholders using the modulo % character.

	● An improved technique uses the .format() string method. ● The newest method, introduced with Python 3.6, uses formatted string literals, called *f-strings*.

modulo % character	print("I'm going to inject %s text here, and %s text here." %('some','more'))
	x, y = 'some', 'more'
	print("I'm going to inject %s text here, and %s text here."%(x,y))
	The general syntax for a format placeholder is
	`%[flags][width][.precision]type`
	`E.g. %5.2f`
	5 would be the minimum number of characters the string should contain; these may be padded with whitespace if the entire number does not have this many digits. Next to this, .2f stands for how many numbers to show past the decimal point.

format() string method	print('This is a string with an {}'.format('insert')) This is a string with an insert print('The {2} {1} {0}'.format('fox','brown','quick')) The quick brown fox print('First Object: {a}, Second Object: {b}, Third Object: {c}'.format(a=1,b='Two',c=12.3))
Alignment, padding and precision with .format()	print('{0:8} | {1:9}'.format('Fruit', 'Quantity')) {0:8} 0 - position of argument 8 - 8 chars if less use whitespaces Fruit | Quantity By default, .format() aligns text to the left, numbers to the right.

You can pass an optional <,^, or > to set a left, center or right alignment:

```
print('{0:<8} | {1:^8} | {2:>8}'.format('Left','Center','Right'))
print('{0:<8} | {1:^8} | {2:>8}'.format(11,22,33))
```

```
Left     |  Center  |    Right
11       |    22    |       33
```

You can precede the aligment operator with a padding character

```
print('{0:=<8} | {1:-^8} | {2:.>8}'.format('Left','Center','Right'))
print('{0:=<8} | {1:-^8} | {2:.>8}'.format(11,22,33))
```

```
Left==== | -Center- | ...Right
11====== | ---22--- | ......33
```

Float precision with the.format() method:

Field widths and float precision are handled in a way similar to placeholders. The following two print statements are equivalent:

```
print('This is my ten-character, two-decimal number:%10.2f' %13.579)
print('This is my ten-character, two-decimal number:{0:10.2f}'.format(13.579))
```

```
This is my ten-character, two-decimal number:     13.58
This is my ten-character, two-decimal number:     13.58
```

Syntax: *{[index]:[width][.precision][type]}*

The type can be used with format codes:

- ***'d'*** *for integers*
- ***'f'*** *for floating-point numbers*
- ***'b'*** *for binary numbers*
- ***'o'*** *for octal numbers*
- ***'x'*** *for octal hexadecimal numbers*
- ***'s'*** *for string*

	- ***‘e’*** *for floating-point in an exponent format* *print('The valueof pi is: {0:1.5f}'.format(3.141592))* *The valueof pi is: 3.14159*
	print('{2} {1} {0}'.format('directions', 'the', 'Read')) Read the directions. —----------------------- print('a: {a}, b: {b}, c: {c}'.format(a = 1, b = 'Two', c = 12.3)) a: 1, b: Two, c: 12.3 —------------------------

Formatting string with F-Strings	name = 'Ele'
	
	print(f"My name is {name}.")

Arithmetic operations using F-strings	print(f"He said his age is {2 * (a + b)}.")
Float Precision using F-strings	print(f"The valueof pi is: {num:{1}.{5}}")
Loops	

While Loop	count = 0
	while (count < 3):
	    count = count + 1
	    print("Hello Geek")

For loop	n = 4
	for i in range(0, n):
	    print(i)

<table>
<tr>
<td>Enumerate</td>
<td>

The `enumerate()` method adds a counter to an iterable and returns it (the enumerate object).

Example

```
languages = ['Python', 'Java', 'JavaScript']

enumerate_prime = enumerate(languages)

# convert enumerate object to list
print(list(enumerate_prime))

# Output: [(0, 'Python'), (1, 'Java'), (2, 'JavaScript')]
```

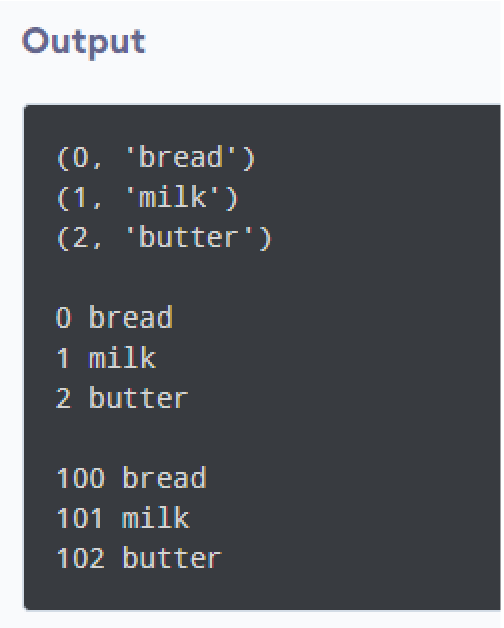
The syntax of `enumerate()` is:

`enumerate(iterable, start=0)`

Example 2: Looping Over an Enumerate object

```
grocery = ['bread', 'milk', 'butter']

for item in enumerate(grocery):
  print(item)

print('\n')

for count, item in enumerate(grocery):
  print(count, item)

print('\n')
# changing default start value
for count, item in enumerate(grocery, 100):
  print(count, item)
```

</td>
</tr>
</table>

	 Output (0, 'bread') (1, 'milk') (2, 'butter') 0 bread 1 milk 2 butter 100 bread 101 milk 102 butter