Python

Resources

The complete guide to Python
Completed!
https://www.youtube.com/watch?v=HGOBQPFzWKo&ab_channel=freeCodeCamp.org

Learn Python With This ONE Project! - YouTube

Basics

Introduction

Python is a high-level, object-oriented and dynamically-typed language with garbage-collection (automatic memory management).

Python is incredibly powerful especially when manipulating and managing data. There is a reason why it is such a popular language with machine learning and big data programmers.

Lines of Execution

The way you see lines differ from how python sees them. For Python, for example, there are no empty lines. And needless whitespace is also ignored. Tabs are treated similarly to whitespace. However, the indentation of a line is incredibly important for python.

Python interprets and handles one logical line at a time.

Python can only handle one logical line on one physical line, it cannot do them at the same time. So you cannot put two print statements on the same line - unless you use a semicolon (;). The semicolon tells python that it is the start of a new logical line.

The opposite of a semicolon is a forward slash (\). It tells python to combine multiple physical lines into a logical line.

Control Flow

Python provides several ways to control the flow of code. In Python, indentation is key to create chunks of code. Everything indented beyond a line belongs to the line, and is subject to its conditions.

Four major ways to determine the flow of code:

- if elif else: run code if conditions are true
- match: 'if' statements for more specific values
- while: repeat code for as long as the condition is true
- for: run code for every item in an iterable

There are commands like break or next to have even more control.

For checking equality, do not use the single equal sign (=), since that is an assignment operator. Use the double equal sign (==) which checks equality without assignment.

Most control flow statements, like the if and while statements, rely on booleans. It will run as long as its input evaluates to True, and stop when it evaluates to False.

Conditions can be combined in order to have more comprehensive control flow
if (condition) -> if (condition1 and condition2) or if (condition1 or condition 2)
if True and True and False or True -> evaluates each one at a time in left to right order.

If statements can be combined and nested to great effect, if needed.

Match Case

Somewhat similar to if - run code if a condition is true. It is better to check one value out of a long list. With match case, declare the variable to check with match and check cases with case.

case _ (underscore) is the equivalent of the else statement in an if-else. It is the catch-all if none of the previous cases are matched.

Match case is really not that much of an improvement over if else. It is simply another option.

if-elif	match-case

if ‘hungry’:
  code
elif ‘tired’:
  code
elif ‘bored’:
  code
	match mood:
	  case ‘hungry’:
	  code
	  case ‘tired’:
	  code
	  case ‘bored’:
	  code

Loops

for i, n in enumerate(iterable)
while (condition)
can break and skip one iteration of the while loop:
break: exits the while loop
continue: skips to the next iteration of the while loop

for element in iterable:
Element can be printed and interacted with in multiple ways

From 0 to a value: use range.
for i in range(value)
Starts from 0, goes up to the value, but does not include it.

A starting value can also be specified in a for loop:
for i in range(start_value, end_value)

Can continue writing on the same line after a colon, but indented code over multiple lines is more readable. But for example, match case lines are often on the same line.

Ternary operator

In the form of
true value **if** condition **else** false value
color = red **if** x < 5 **else** color = blue

This is a very readable and efficient way to create single line if-else statements. This would work in a lot of different tools as well. For example, you can use the ternary operator inside an f-string, a list, and even a function.

Documenting Functions

Comments

Comments don't influence the code, they just explain. Good form of communication, vital portion of all good code. Helps with organization as well.
Create line comment: #, only works on one line
Create block comment: ''' (body) ''', will work over multiple lines

Command + forward slash = comments / uncomments all selected lines of code

Docstrings

Docstrings are comments that summarize a body of code. It usually also includes the expected types of data for parameters and the return value.

Always use block comments for the docstring.

Printing a function with docstring: print(func.__doc__) or help(func)

Data Types Intro

Data type	Notes
string	Immutable in reality - a new string is constructed with any modification to an existing string. The variables that strings are commonly assigned to, however, are mutable. That is why string concatenation exists. char could be considered to be a string with length one

int	No decimal places
float	Handles decimal places
list	A mutable container to store other data types Default is shallow copy, using pointers
tuple	An immutable container to store other data types
set	A list that cannot contain duplicate values
dict	A mutable container that contains key: value pairs
bool	Can only ever take on one of two states: True or False.

In general, python is incredibly relaxed about data types. It can combine data types flawlessly to the point that you do not really notice. For example, python changes datatypes easily.
- Adding an int to a float will produce a float.
- Performing division will always produce a float, even if it is not necessarily needed.
- functions that change data types such as int(), float(), str(), bool(), list(), dict(), tuple(), set() are very forgiving.

There are tons of additional data types that are not explicitly used.
None - the absence of a value
Sequence - to get a range of numbers
Bytes, complex numbers, memoryview, frozensets - very specific types of data for highly specific purposes.

Python also has a tone of functionality that converts data.

Data conversion function	Notes
Type()	Returns the type (class) of the input
float(param)	
int(param)	Truncates values beyond the decimal point,

	not round

Variables
A variable is a container for any kind of data.

Variable naming conventions:

- Can only contain letters, numbers, and the underscore symbol. No $, %, or space.
- Must start with a letter or an underscore (not a number)
- Must be different from the inbuilt python tools (such as print)
- Use snake_case
- Should make sense and be consistent

To efficiently compute mathematical operations between variables, add the '=' operator after a math operator. For example,
A += B will add the value of B to A.

Mathematical Operators

A (operator) B

Operator	Meaning	Notes
+	Addition	
-	Subtraction	
*	Multiplication	
/	Division	
**	Exponentiation	
//	Floor	Truncates all values of A past the decimal point
%	Modulo	Returns the remainder of the division
()	Parenthesis	Takes priority during order of operations

Strings

Strings in python are iterable but immutable. They can be concatenated and multiplied.

Both single (‘) and double (“) can be used during string construction, but do not mix the two - for example, “test’ would be invalid.

Quotes inside quotation marks: either mix quote types (double quotes in a single quote string) or use the simple escape character (\). Python will interpret the symbol immediately following the simple escape character as a string. ‘\”\”’ will be a string that prints out a double quote and then a single quote.

Newline escape character: \n. Can also create a variable with multiple lines by using the triple single quote (‘’’) on each side of a string declaration.

When printing strings, use fstrings:
print(player_name) -> `print(f'Your name is, {player_name}')`
The alternative is the format string, whose unwieldy nature is why f strings were invented in the first place:
‘Hello {}, you are {} years old.’.format(name, age)’

Strings are extremely similar to lists and tuples. There are many methods to easily convert python strings to lists and tuples.

String conversion Method	Notes
.split()	
list(param)	Converts a string param into a list
tuple(param_string)	Converts a string param into a tuple
sep.join(param)	Joins a list / tuple param into a string with the separator ‘sep’ between each element. Most of the time the sep would be a space (‘ ‘), and it is vital that each element in the input is a

	string.

Lists and Tuples

Lists are a data type that uses consecutive memory in storage to store information. It can be used as a stack, a queue, a tree, or a graph. They are defined with square brackets. There are tons of methods to use on lists.

List Method	Notes
append(param)	Adds the element at the end of the list
reverse()	Reverses the elements in the list

Tuples are effectively immutable lists. Tuples may contain any type of data, including tuples themselves. External modifications to mutable elements inside a tuple will update the tuple despite its immutable nature due to the pointer-oriented nature of python code. Tuples are defined with (normal) brackets.

Both lists and tuples can be unpacked - multiple variables can be assigned to elements of a list or tuple. It is key to have the same number of elements as variables.
a, b = (10, 15)

Indexing and Slicing

Elements in a list and tuple are ordered starting from index 0 as the first element. We can access the elements in a list and tuple using square brackets with the desired index number in between. This action, called indexing, can be used on most iterables, including lists, tuples and strings. It does not work on dictionaries and sets. Negative values can also be used for indexing, where the last element in the container is -1.
my_list[1]

To index multiple values at the same time, python supplies a feature called slicing. Slicing involves square brackets that contain three numbers separated by a colon. The first is the start, the second the end.
my_list[start:end:step]

Importantly, slicing goes up to the end but does not include it. Slicing would return 1 less element than you think it would.

By default, the step size is 1, the start value is 0 (the first value of the list), and the end is -1 (the last value in the list).

Most iterables, including lists, tuples and strings, can be sliced.

Dictionaries

Dictionaries are python's interpretation of hashmaps. They are defined by key: value pairs separated by commas in curly brackets. Adding a key: value pair to a dictionary that already contains a pair with the same key will result in the old value being overwritten. Values of the dictionary can be directly accessed by dict[key]. Key: value pairs can be added by dict[key] = value.

They are exceedingly powerful in so many situations that a common mantra of competitive programming is as follows: when in doubt, dictionary. Its quick, O(1) access and update time, combined with its intuitive and simple structure, makes it one of the most versatile data storage types in all of computer science. Dictionaries can also be nested to create powerful representations of hierarchical data.

```
bank = {'jack': 4098, 'sape': 4139}
print(bank['jack'])
bank['john'] = 600
del bank['sape']
for name, balance in bank.items():
   print(f"{name} has a balance of {balance}.")
```

Each key must be unique and immutable. Converting a dict to a list or tuple (using the list() or tuple() methods) will create an appropriate container containing all of the keys of the dictionary. Converting a dict to a string will result in the entire dict as a string, including the keys and the values.

There are many useful methods for dictionaries in python.

Dictionary Methods	Notes

.items()	Returns a list containing a tuple for each key: value pair
.keys()	Returns a list containing the dictionary's keys
.pop(param)	Removes the element with the specified key
.values()	Returns a list of all the values in the dictionary
.get()	Allows access of an element with a failsafe that returns a default value if the key was not found, mitigating keyError exceptions
.update(param)	Param usually in the form {key: value}, adds the pair to the dictionary, overwriting the old value if it existed.

Sets

Sets are less like lists without duplicate values and more like dictionaries without values. Each of its elements must be unique or else it will be deleted. A set always has curly brackets, much like dictionaries.
{ elem1, elem2, elem3, elem4 }

It is important to note that indexing and slicing on sets does not work. In fact, there is no real easy way to select one specific element in a set. The closest method is .pop(), which removes the first element from the set and returns it.

Sets are very good when it comes to comparisons. .union, .intersection, and more, are all mightily useful. But sets as a whole are a niche data structure, much rarer than dicts, lists and tuples.

As is common in python, a set can be constructed from a list with the set(param) function.

There are many useful methods for sets in python.

Set Methods	Notes

.add(param)	Adds the param to the end of the set
.remove(param)	Removes the item from the set
.pop()	Removes the first element from the set and returns it.
.union(param)	Creates a new set with all involved elements Can also be written as (set 1 | set 2)
.intersection(param)	Creates a new set with shared elements Can also be written as (set 1 & set 2)
.difference(param)	Creates a new set with elements only present in the original set Can also be written as (set 1 - set 2)

Booleans

Deceptively simple, can be used to great effect in control flow of code and general logic. Booleans are most commonly used while using comparison operators.

Comparison Operator	Notes
==	Is equal
!=	Not equal
<, <=	Smaller than, smaller or equal than
>, >=	Larger than, larger or equal than

not reverses a boolean. not true is false, and not false is true. Booleans can be used to check, for example, if there is a value in a list or string.
ele in container -> returns a boolean

Bool(), Truthy and Falsy

bool() can accept any number, string, type of container and still return a value. This makes things very complicated since data can be converted in so many ways.

Truthy - values that are converted to True
Falsy - values that are converted to False
Relevant falsy values include:

- 0 or 0.0 (int and float)
- '' (empty string)
- [], (), {} (empty list, tuple, set, dict)
- None (absence of a value)

Anything else is truthy.

Functions

A function is simply a block of code that can be reused, and always gives the same output given the same input (unless it is intentionally made random). Much like other control flow tools, we indent the body of the function. When the function is used, that is called 'calling' the function.

Inside the brackets we can add parameters. Multiple arguments are separated by commas. abs() is an example of a function that can only take in one argument, while max() can have multiple arguments.

function(param1, param2, param3)

pass can be used to fill function bodies temporarily. It has no behavior.

To see what functions really are in python, see the OOP and Decorations sections.

Scope

It is key to note that variables created inside of a function are only available inside of that function. This 'local scope' limits the usage of those variables outside of the function, the 'global scope'.

Every function has its own local scope and every local scope is separate. For example, if you declare a variable in a function with the same name as a variable in the global scope, python will only interact with the local variable in the function body. This also applies to different functions, so you can use the same variable names for every function.

You can access global variables inside functions, but you cannot update them. You can move between scopes with parameters. Use global and return when needed. Return is much preferred over global.

- Global allows you to declare the interpretation of a variable into its global state, so you can access them inside functions.
- Return allows you to return values from functions back into the global scope. Parameters allow you to pass values from global into the function. This forms a completed cycle.

Commonly used built-in functions:

Built-In Function	Notes
print()	
len()	
abs()	
max()	

Method

Methods are functions that are always attached to objects:

object.method(argument)

Sometimes functions only make sense when they are attached to data types or objects. For example, upper() is a method because it would only make sense on letters.

Functions can be nested. The return value of one function would effectively serve as the argument of the next.

Commonly used methods:

Method	Used On	Notes

upper()	strings	Turns all letters to uppercase
strip()	strings	Removes starting and ending whitespace
title()	strings	Capitalizes the first letter of every word
replace(old, new[, count])	strings	New substring replaces old substring for the first count occurrences; if count is not specified, all instances are replaced

Return

Return is the glue that connects all functions. All operations return values, and functions and methods are no exception. It is why nested functions work, how calculations and adjustments to anything work, and so much more.

Every operation returns some type of value. Understanding this is supremely important, especially when code grows more complicated.

Parameters and Keywords

Arguments are assigned to parameters by position by default. But you can also specify the desired parameter to be more precise.

A convention is that position arguments must come before keyword arguments.

To handle situations where you do not know the number of arguments, you can simply use the parameter ‘arguments’, which will be a list.

func(arguments):

But python also has a special way of dealing with this multiple situation - list unpacking. Add a star before the arguments keyword

func(*arguments):

This would create a tuple called arguments with every element inside of it.

The arguments keyword can coexist with previous parameter inputs, it will simply take all of the inputs not absorbed by a previous parameter.

Finally there is keyword unpacking:

func(**arguments):

With the double star, ‘arguments’ becomes a dictionary with keyword: value pairs.

We can combine both list unpacking and keyword unpacking:

func(*args, **kwargs):

This simple and elegant parameter list will take all non-keyworded arguments in a list called argos, and take all keyword: value pairs in a dictionary called kwargs.

On the flip side, parameters can be optional by assigning to it a default value.

func(first, num = 7):

You can specify the desired type of the input using a colon:

func(first: string, num: int = 7):

To specify the type of output, use an arrow BEFORE the ending colon of the function:

func(first: str, num:int = 7) -> int:

Note that these are all optional. Even if they are violated, the function could still work.

Lambdas

Lambdas are just single line functions. These simple functions are very useful in situations where you would like to transform elements at the last minute if they meet a condition.

lambda parameter: expression
lambda x: x + 1

The result of the expression is automatically returned. For the example above, if 10 is the input, it will automatically return 11.

A lambda function can be assigned to a variable, and you can then treat the variable as a function (complete with the parenthesis)

Some functions require or allow the use of lambdas. For example, the sort() function.
sort(list, lambda)

Example lambda function with the ternary operator:

a = lamba x: ‘hello’ if x > 5 else ‘bye’

File Handling

By default, python can open simple files like .txt files. Python can open much more sophisticated file types with the assistance of external models.

Manually:
file = open(‘file_name, ‘access_mode’)
Multiple access modes: r, r+, w, w+, a, a+, b # read, write, append
file.close()

Be sure to close the file when it is no longer needed. It will free up memory from runtime.

Automatically:
with open(‘file_name’, ‘access_mode’) as file:
    [can only interact with indented code]

Closes automatically after leaving the scope. Think of it as an indented special local scope - like an if or while.

Interacting with the file:
f.read(n) # reads the first n contents of the file

for line in list(file):
f.readline()
f.write(str)
f.writelines(str)

If you input a nonexistent file with write mode, python will create a new file with that name.

Iterators

Iterators are objects that iterate over iterable objects. An iterable object is one that can return one of their elements at a time, such as lists, tuples, and strings.

```
while True:
   iterator = iter(iterable)
   element = next(iterator)
   print(element)
```

In python, the StopIteration exception is raised when an iterator runs out of steam. There is no graceful control like the comparisons with the end of iterable as seen in C++. This is because iterators are usually used implicitly in python.

Shallow vs Deep Copies

A shallow copy creates a new object but does not create copies of nested objects, instead it just uses references to those nested objects.

A deep copy creates a new object and recursively copies all the objects it contains.

In python, the assignment operator does not create a copy at all, merely just binding a name to the existing object.
Shallow copies can be created using methods like copy()
Deep copies can be created using methods like deepcopy()

Data Manipulation Techniques

Zip, Enum

Thing	Notes
zip(list1, list2)	Combines / merges the elements of the two lists pairwise; the first element of the first list will be in a tuple with the first element of the second list, the second element of the first list in a tuple with the second element of the second list, and so on. If we try to print it out directly: prints out a definition and a memory, since there is no defined print for a zip object. After converting to a list using list(): [(a[0], b[0]), (a[1], b[1]), ...] If we iterate through zip, one tuple will be returned at a time
enumerate(iterable)	Enumerate returns a tuple at each iteration. The tuple contains the index number and the element at that index. Normally used inside a for loop: for index, name in enumerate(iter_list):

List Comprehensions

List comprehensions allow the creation of a list using just one line of code. It is one of hte most powerful features in python, being able to create new lists and manipulate existing lists with ease. Combined with a ternary operator, they become truly powerful, especially at filtering or modifying other lists.

lst = [(modified_x) for x in (iterable)]
lst = [(true_x) if statement else (false_x) for x in (iterable)]

Examples of basic list comprehensions:
The squares of 0 to 9

```
squares = [x ** 2 for x in range(10)]
```

The numbers from 0 to 99

```
linear = [x for x in range(100)]
```

The numbers from 0 to 99, except each element is repeated thrice in a tuple:

```
linear_triple = [(x, x, x) for x in range(100)]
```

Examples of ternary list comprehensions:
The numbers from 0 to 99, excluding numbers below 10:

```
linear_selective = [num if num > 10 else 0 for num in range(0, 100)]
```

Used to modify other lists:

```
Replenish_list = [(name, number) for name, number in zip(inventory_names, inventory_numbers) if number < 25]
```

List comprehensions can be combined.

```
combined_comp = [[(x, y) for x in range(5)] for y in range(10)]
```

This is incredibly powerful, for example. combined_comp is now a matrix with tuples as elements, where each 0th index value is equal to its column number and each 1st index value is equal to its row number.

List comprehensions can easily create the fields of a chess board:

```
chess_board = [[f'{letter}{num}' for num in range(1, 9)] for letter in 'ABCDEFGH'[::-1]]
```

Other comprehensions

List comprehensions are not the only comprehensions. There exists at least a pseudo-comprehension for almost every container in python.

A dict set comprehension looks as follows:

```
dict_comp = (key: value for x in iterable]
dict_comp = (num: num for num in range(10)]
```

A set comprehension is the exact same as a list comprehension, but you replace the square brackets with the curly brackets.

```
set_comp = {num for num in range(10)}
```

A tuple comprehension is the same as a list comprehension, except with the tuple() function.

```
tuple_comp = tuple(num for num in range(10))
```

This will create a tuple with 10 elements. Tuple comprehensions are extraordinarily rare.

dict_comp = (x: [y for y in range(6)] for x in 'ABCDE')

Sorted()

sorted() is an excellent function for sorting data, and takes a function as an optional argument.

sorted(iterable, function)

It has a keyword argument ‘reverse’, which can be set to true:
print(sorted(list1, reverse = True)

The keyword argument that allows you to pass in the function is called ‘key’.

print(sorted(list2, key = sort_function))

Note to not call the sort_function, so do not include any brackets right after its name.
Sort_function needs to be a function that returns an integer, and python uses it to order the input list.

In reality, you rarely need to make dedicated, complicated functions. Instead, you use lambda functions.

Sorts a list with tuples by the values of their second elements:
print(sorted(list2, key = lambda item: item[1])

Map(), Filter()

map(function, list) creates a new list by passing in the values of a list iteratively through a given function.

filter(function, list) filters out values based on a boolean function. The output of filter() needs to be manually converted to a list.

Usually lambdas are used instead of entire functions. Both map() and filter() have been mostly replaced by list comprehensions.

Good-To-Know

Del, Pop, Clear

Since python is inherently reference-based, we can delete variables using the del keyword.

Delete directly:
a = 1
del a
print(a) # error

Delete by index:
a = [1, 2, 3]
del a[1]
print(a) # [1, 3]

Removes an item by value:
a.remove(3)
print(a)

In python, since memory is automatically allocated and deallocated, there is not much need to use del manually.

Pop() removes the specified item by index and returns its value. The default is -1, the last item in the container.

clear() clears the entire list, making it empty.

Object Oriented Programming

Object oriented programming is a code design philosophy used by almost all large projects in any coding language. The four pillars of OOP help greatly with code maintenance and reuse.

The Four Pillars of OOP

The four pillars of OOP are:

- Encapsulation: each object should control its own state. Instead of direct access, use data hiding and setters. Getters should also be used.
- Abstraction: make classes and functions easy to use with detail hiding.
- Inheritance: relatives (children) of an object should acquire the properties and methods of another object. A child class should have all the functionality of its parent class, and if used in its place, it should perform the same. This means that no overridden method should change expected behavior.
- Polymorphism: methods should do different things based on the object it's called on. This is typically achieved through method overriding.

All pillars promote code reuse.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def eat(self):
        return f"{self.name} is eating."

    def sleep(self):
        return f"{self.name} is sleeping."

class Human(Animal):
    def think(self):
        return f"{self.name} is thinking."

animal = Animal("Leo the Lion")
human = Human("Alice")

print(animal.eat())    # Leo the Lion is eating
print(animal.sleep()) # Leo the Lion is sleeping

print(human.eat())     # Alice is eating (inherited)
print(human.sleep())   # Alice is sleeping (inherited)
print(human.think())   # Alice is thinking (specific to Human)
```

Objects and Classes

An object is a container for variables and functions, molded from a class, which provides the blueprint and functionality of the object. For example, an object called a monster.

- Variables in an object are called attributes (health, speed, damage, energy, etc)
- Functions in an object are called methods (attack, growl, feed). Class methods always need at least one parameter, a reference to an object. By convention, that parameter is called self.

It is possible to create multiple objects all stemming from a blueprint class. Each object has its own attributes and methods that apply to their own respective object.

Objects can interact with each other. One monster object can attack or interact with another monster object.

A class is the blueprint for an object, so when creating an object, we first need a class. A class also accepts arguments to customize the object. For example, a class called Monster, and an object created from it called shark.

Even if you dislike OOP, it is unavoidable.

1. They organize complex code
2. They help to create reusable code
3. They are used everywhere
4. Some modules require you to create classes
5. They make it easier to work with scope

Most people avoid using OOP as a programmer, but they should practice from early on.

A class can also inherit from another class. The resulting objects will have attributes and methods from both classes. For example, passive monsters, armored monsters, speed monsters - they can all be child classes of the Monster class, which itself might inherit from the Entity class.

Functions are Objects

Finally, we have to realize that everything in python is an object - including the inbuilt strings, integers, etc. Even functions are objects! A function object just uses its __call__ method whenever it is called with parenthesis.

That means that you can store functions in variables, since functions are just objects. **It is even possible to pass in functions into classes to turn it into a method**:

```
def add(a, b):
   return a + b

class Test:
   def __init__(self, add_function):
       self.add_function = add_function

test = Test(add_function = add)
```

```
print(test.add_function(1, 2))
```

All that happened was that you gave the class an object with a declared __call__ method, which it then used to construct an object with the same __call__ method. And since class methods are just functions but for classes, this seamlessly converts an outside function into a class method. Be sure to not use parentheses when passing in functions as objects.

Scopes and Classes

Since every method has a reference to the class, it is easy to get and change class attributes. Because of that, methods rely much less on parameters, global and return.

Objects can be influenced from the outside and from a local scope of a function. This prevents many problems with scope that makes things very complicated.

Simple Inheritance

Inheritance simply means that one class gets the attributes and methods from another class (or classes). The child class inherits from the parent class. Although extremely powerful for code reuse, be sure to not go overboard with using inheritance in relatively simple code, when composition could do the job instead.

A class can inherit from an unlimited number of other classes, and a class can be inherited from by an unlimited number of classes.

Declare an inheritance relationship as follows:
class Child(Parent):

And in the __init__ dunder method, we can call the parents' __init__:

```
class Shark(Monster):
    def __init__(self, speed, health, energy):
        Monster.__init__(self, health, energy)
        self.speed = speed
```

Calling the parents' __init__ has now been replaced by calling super(), as follows:

```
class Shark(Monster):
    def __init__(self, speed, health, energy):
        super().__init__(health, energy)
        self.speed = speed
```

super() has replaced ‘raw calling’ the __init__ of the parent class due to its much more versatile nature, especially in multiple inheritance.

Complex Inheritance

Create a child class with multiple inheritance by inserting more parent classes between the parentheses. In order to use super() correctly in this case, one needs to understand the MRO - the method resolution order. The order which the involved classes are being called in the constructor.

```
print(Shark.mro)->
[<class '__main__.Shark'>, <class '__main__.Monster'>, <class '__main__.Fish'>,
<class 'object'>]
```

With the MRO, we know to first input parameters for Monster in the first super() and the parameters for Fish in the second super(). It is typically in the order declared in the class definition.

```
class Shark(Monster, Fish):
```

Self -> Monster -> Fish

Think of multiple inheritance in python as a chain that follows the order as given by a class’s MRO. We first initiate self (class #1), and ‘hop’ to the next class (class #2) with super(), while passing in the necessary parameters. We then ‘hop’ to class #3 with super() in class #2’s __init__ after the setup for class #2 itself is done. This is incredibly unwieldy because in order for class #3 to receive the necessary parameters for setup, both class #1 and class #2 needs to modify their __init__s to contain the parameters that are ultimately only useful for class #3. This is multiple inheritance in python.

An alternative to passing every parameter through every class is to use keyword unpacking with **kwargs.

Public vs Private attributes

Public attributes can be accessed and changed outside of the class, while private attributes cannot. Private attributes are marked with an underscore before the formal start of the name. For example, the attribute ‘id’ would be ‘_id’ if privatized.

Others

hasattr(object, ‘attribute’) checks if the object has an attribute. It is useful, especially inside of if statements.

setattr(object, ‘attribute’, ‘new_state’) sets the specified attribute of the object as new_state. This allows for very efficient ways to create new attributes for an object. Other than that, it is just another way of stating object.attribute = new_state.

__doc__ is a method that returns the docstring of the object.

Dunder Methods

__Dunder__ methods (aka double underscore methods) are methods that are not called by the user - instead, it is called by python when something happens.

__init__ is called when an object is created
__len__ is called when the object is passed into len()
__asbs__ is called when the object is passed into abs()

Classes need dunder methods such as init in order to achieve customizability. __init__ allows passing in parameters to customize the class.

```
class Monster:
    def __init__(self, health, energy):
        self.health = health
        self.energy = energy

    def attack(self, amount):
        print('The monster has attacked!')
        print(f'{amount} damage was dealt!')
        self.energy -= 20
        print(self.energy)

    def move(self, speed):
        print('The monster has moved')
        print(f'It has a speed of {speed}!')
```

By using the __init__ dunder method, we forgo the need to declare all attributes manually. Simply declare them in __init__.

You can also declare the __len__ and __abs__ dunder methods in a class, which can be implemented much akin to operator overloading in C++.

The __dict__ dunder method returns a dictionary with attribute: value pairs of that specific object. It is called without parenthesis:

```
print(monster.__dict__)
```

The __call__ dunder method basically turns the class into a function. For example, monster could then be called as monster(), and could also have parameters.

The classic C++ operator overloads are implemented as dunder methods in python, such as __add__, which allows you to directly combine an object with the class with anything else, as long as it is handled correctly.

There are a variety of other dunder methods, such as __str__, which handles behavior when str() is used on the object.

Modules and Outside Help

Modules

Modules are extra parts that can be imported in order to assist with programming. They can be used as tools, libraries, classes, and more.
Importing modules from the standard library: import (name), no further action needed

Standard library modules	Notes
random	random.randrange(a, b) random.randint(a, b) random.choice(seq) Note that randint(a, b) is an alias for randrange(a, b + 1).
math	math.floor(a)
datetime	datetime.time() datetime.now(tz = None)

time	time.sleep()
pyautogui	pyautogui.write(msg, interval = time)
Pyplot matplotlib	Makes good graphs.

Importing modules outside of the standard library: download onto computer first, import with correct directory

If you are just looking for specific commands from a module, do
from (module) import (function) as (name)
And then the specification of the module would no longer be necessary. If floor is imported from math, you can call floor() directly.

External modules are made by other programmers and provide a huge amount of extra functionality. The only difference between an external module versus a standard library module is that you have to download the module on your computer first.

Using pip to install supported modules:
pip install (module)
pip uninstall (module)

Creating Modules

Creating modules helps with organization. Each module is in a separate file and this multi file compile allows you to keep file sizes manageable.

Create a new py file (which will become the module). Import functions in that file to other files using
import (file_name)
And then use those functions by declaring
file_name.function() whenever function() is supposed to be used.

It takes wisdom to realize the power of having multiple files, all with minimal code inside. Makes organization and debugging much easier.

Dunder Main

When a python file is called, it creates a few interval variables. The most used one is __name__, which gives a name to the currently executed file.
The currently executed file
__name__ == '__main__'
Any imported file
__name__ == '__filename__'

This helps us control what code is executed. The currently executed 'main' file is always going to have __name__ == '__main__', while any imported files will have varying names depending on their file name.

That is why we always have a if __name__ == '__main__':. This helps us confirm that we are running the current file as a main file.

User Input

Input() can be used to receive user input. It can deliver a message like a pseudo-print.
user_input = input('message")

Error Raising and Handling

Error Handling

Errors are sometimes unavoidable.

```
try:
        (risky code)
except (error_type):
        (behavior)
except (error_type_two):
        (behavior)
else:
```

An except without specifying an error type will be a catch-all, although that would be less helpful during the debugging process. Except statements can be stacked, much like else and case statements. Finally, after all excepts, we can have an else statement that runs when the try statement does not have an error.

Common Errors	Notes
ZeroDivisionError	When dividing by 0 is attempted
NameError	When the variable is unrecognized
AssertionError	When an assertion statement evaluates to false
IndexError	When an index for a container could not be found

Raising Exceptions

It is possible to raise exceptions ourselves.

```
var_must_be_string = 'test_string'

if isinstance(var_must_be_string, str):
        print(var_must_be_string)
else:
        raise error_type('Must be a string')
```

Assert

Assert is a stronger if statement. It will only allow code to run if its condition is true.
assert (bool)
If the bool evaluates to False, assert will raise an error - AssertionError. This is especially useful for cases when security matters.

Decorators

Decorators are functions that 'decorate' other functions - we essentially wrap a function around another function.

The desired function is called during the middle of the decorator, allowing code to be executed before and after the main function. This allows us to give a function extra functionality without changing it.

Decorators are commonly used to

- Test code without changing it
- Avoid unnecessary code changes
- Run code when an attribute in a class is accessed or changed

Decorators leverage the fact that uncalled functions are, as discussed in the OOP section, just an object.

For example, there exists functions that can create function objects that can then be returned and used out of scope.

```
def func():
    pass

def wrapper(func):
    print('hello')
    func()
    print('goodbye')


def function_generator():
    def new_function():
        print('New Function')
    return new_function

new_func = function_generator()
new_func()
```

Decorators can permanently alter any function by declaring a new function (a wrapper) that includes the old function and surrounds it with new code. The decorator function would then return that altered function, which would replace the input function entirely.

```

def decorator(func):
    def wrapper():
        print('decoration begins')
        func()
        print('decorator ends')
    return wrapper

def func():
    print('function')

new_func = decorator(func)
new_func()
```

Eval and Exec

Eval and exec are special functions that translate strings into python code. For example, Eval(‘1 + 1’) returns 2. It can even be used to create new variables. But you need to be extremely careful when using them. Or else, it allows users to use their own code in your code.

Eval: less powerful, can only execute code. Can run functions and simple operations but cannot create any new variables.

Exec: is more powerful, can execute any code, including the creation of variables.

Random

Composition vs Inheritance

Inheritance: subclass that extends a class’s functionality, with new methods and overrides. Inheritance handles code reuse by providing a way to extend the functionality of a parent class, and handles abstraction using different layers of classes and the parent-child class separation.

Downside: limited flexibility due to tight coupling between parent and child classes. The child class may inherit methods and parameters that do not make sense for its context, requiring change for good design. Change is the enemy of perfect design, and inheritance is no exception. So changing the child class would require changing all the classes in order to not break expectations. This is a highly costly endeavor.

Composition: simply reusing (copy pasting) the code that multiple classes desire. There are no dependencies between classes since they are independent from one another. Handles code reuse by just using the code. And handles abstraction using interfaces, a contract describing what an object can do.

Composition also gives rise to dependency injection. Major cons of composition is its (obvious) code repetition and wrapper methods (useless getters).

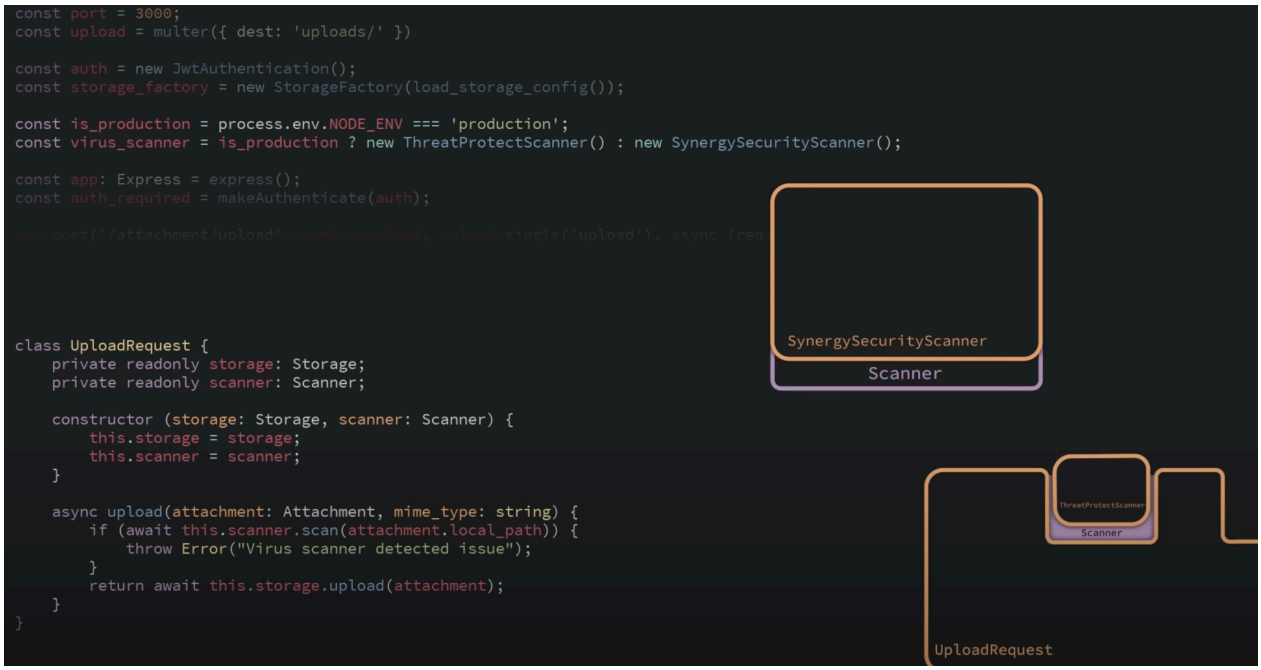
Inheritance is not evil. Sometimes it is a good idea to use inheritance. When using inheritance, design the parent class to be inherited (avoid protected member variables with direct access), create a protected API for child classes to use.

Dependency Injection

Dependency injection is when a piece of code uses another piece of code, except instead of using the other piece of code directly, it is passed in as an argument. When something is passed in to be used, it is called injection.

Consider a scenario where a user's payload might need to arrive at different servers depending on the company the user is affiliated with. Extending if statements is the intuitive option, but it is slow and will have a lot of unnecessary complexities, making calling difficult. If we use the principles behind dependency injection, we can instead first define the payload by defining it on the top and then just passing the defined parameter into all subsequent functions.

Basically the principle is similar to the encapsulation and abstraction principles of OOP. If we have multiple potential 'final states' of a variable, we define it first abstractly and then pass it along to all subsequent functions. That way changing code is extremely simple, if for example we want to switch to another state.



The Downsides of Abstraction

Abstraction Can Make Your Code Worse

With increased abstraction comes increased code coupling. It makes modifying parent classes and even child classes extremely difficult and dangerous, due to how intermingled they are.
Code repetition is not always bad, and abstraction sometimes provides no value.
Coupling means constraining two or more classes to the same parent class or method, making them much less flexible and more resistant to change.
It's only worth it to use abstraction when its benefits outweigh the consequences of coupling.

Tech Stacks

How to OVER Engineer a Website // What is a Tech Stack?
Frontend, APIs, Backend
APIs are not just the connectors of the frontend and backend, but also can include anything that would be too difficult to implement from scratch (like payment, user authentication, texting, and image recognition)