Python - Beginner to Advanced

The Python Beginner to Advanced cheatsheet is a comprehensive reference guide that covers the most important concepts and syntax of the Python programming language. The cheatsheet covers a wide range of topics, including data types, control structures, functions, modules, classes, objects, file handling, regular expressions, and more. It provides concise and easy-to-understand explanations of key concepts, along with examples of their usage.

With this cheatsheet, developers can quickly find the information they need and improve their productivity when working with Python code.

Contents

Numbers and Math

Numbers

- `float` uses more space than `int`
- There is only 2 no. b.w 0-1 but there are infinite nos. b/w 0.0 and 1.0
- Adding `int` and `float` always returns `float`
- Use `type()` func to check the type
- `1/2` will return a float `0.5` but a lot of other lang ignore the decimal and don't use float
- Operators follow the hierarchy rule called PEMDAS or BODMAS
- `4**3` = `4*4*4`
- `49**0.5` = `7` as it takes the sqrt
- `25%4` = 1 -> it gives remainder and used to find if num is even or odd
- `10/3` = 3.3333333 -> returns a float
- `10//3` = 3 -> returns just an int

Comments

- Help us to write notes inside our code

Variable and Strings

Variables

- Store values/data to access them later
- They can store all sorts of data types
- They need to be assigned before they can be used
- Variables can be assigned to other variables

```
 a = 10
b = a
```

- Can be reassigned at any time

```
 a = 10
a = 50
```

- Can be assigned at the same time as other variables

```
  a, b, c = 10, 25, 50
```

Rules for naming Variables

- have to start with letter or underscore only
- name must contain letters, numbers, or underscores
- names are case sensitive

Naming Conventions for Variables

- should be named using snake_case
- usually should be lowercase
- CAPITAL_CAMEL_CASE refers to constants like pi(3.14)
- UpperCamelCase refers to classes
- variables with double underscore should be left alone and shouldn't be messed with - `__no_touch__`

Data Types

- any assigned values must be a valid data type
- there are many `int`, `float`, `str`, `boolean`, `list`, `dict` and much more

Data Type Conversion

- data types can be converted from one form to another

```
 decimal = 3151.15155
integer = int(decimal)
# integer now equals to 3151
```

Strings

- any character, symbol, word or letter
- any valid Unicode character

```
some_string = 'My name is Durgesh'
```

- can be declared with either single or double quotes (just try to stay consistent)

Escape Sequences

- special sequences used inside strings to do special task via python
- can be used to escape quotes when we have quotes inside quotes
- many more in the docs

String Concatenation

- used to combine multiple strings together
- can be done using `+` operator in python

Formatted Strings

- called F-strings, used to interpolate things
- converts `int` into strings when written inside unlike concatenation

String Indexes

- can be used to access a particular character from a string using `[]`
- indexes are 0 based
- can be accessed using negative numbers as well

Dynamic Typing vs Statically Typed

Python is dynamically typed language. You can reassign a variable to different data types which isn't the case for many other languages.

C++, for example is statically typed where if we define a variable as boolean, it has to stay boolean otherwise there will be an error

None

- Python's version of **null**
- Represents nothingness
- Used when we want a variable declared but don't have anything to assign to it just yet

Booleans and Conditionals

Input Function

- Gets an input from users
- Built-in function
- Result is always a string
- a msg can be passed inside for the user

Conditional Statements (If-Else)

- use `if-else` statements to make the program do different things based on some logic/condition
- we can do a lot more than just print stuff inside `if-else`
- we can avoid elif and else statements too based on our needs
- we will only have one `if` and one `else` but we can have as many `elif`'s as we want

Comparison Operators

- there are many comparison operators - `>, <, >=, <=, !=, ==`
- all of the comparison operators return True or False

Truthiness and Falseness

- All conditionals resolve True or False
- False values are - empty objects, empty strings, None, and zero

Logical Operators

- used to combine comparison operators to create boolean logic
- 3 types - `AND, OR, NOT`
- AND - Truthy when all the conditions are true
- OR - Truthy when any one of the conditions are true
- NOT - Truthy when opposite of the given condition is true

is vs. "=="

- they are very similar too

  ```
   a = 1
  a == 1 #true
  a is 1 #true
  ```

- They are different as `is` compares the instance and not the value

  ```
   a = [1, 2, 3]
  b = [1, 2, 3]

  a == b # True (same value?, yes)
  a is b # False (same instance?, no)

  c = b
  c is b # True (they're both pointing to the same thing in memory)
  ```

- `is` is only true when 2 values reference the same item in memory

Loops

- used to iterate over things
- we have iterable objects or data and we run the loop for every item in that collection

While Loop

- runs as long as a value is truthy
- conditional statement runs only once if a condition is true but this keeps running as long as the condition doesn't become falsy
- BE CAREFUL AND DON'T YOU DARE CREATE AN ENDLESS LOOP

Lists

- are data structures to store other different data type collections
- there are a whole lot of methods for lists

List Methods

- `append` - inserts a new item at the end
- `extend` - inserts the given items(or a new list) at the end and extends the original one
- `insert` - inserts an item at the specified index
- `clear` - removes all the items from the list
- `pop` - removes and returns an item from the given index (by default removes from the last)
- `remove(x)` - removes the first instance of the given item
- `index(x)` - returns the index of the specified item
- `count` - returns how many times an item occurs in a list
- `reverse` - reverses the list w/o creating a new one
- `sort` - sorts item based on asc order
- `join` - technically a string method but is used to combine/concatenate items of a list

  ```
   words = ["Coding", "is", "fun!"]
  text = " ".join(words) # joins the words with a space
  print(text)
  ```

List Comprehension

- does a specific task for every iteration when looping over a list and it creates a new list

  ```
   nums = [1, 2, 3]
  # create a new list where the items are doubles
  doubled_nums = [num*2  for num in nums]
  print(doubled_nums)
  ```

- technically, we can do the same using loops but that'd be a bit tedious
- used a lot with data science, web deb, etc.

LS with Conditional Logic

```
 numbers = [1,2,3,4,5,6]

# store even values
evens = [num for num in numbers if num % 2 == 0] # [2,4,6]

# if even, double it and if odd, half it
weird = [num*2 if num%2==0 else num/2 for num in numbers] # [.5, 4, 1.5, 8, 2.5, 12]
```

Nested Lists

- lists inside lists
- used a lot in almost all of the fields

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

  print(nested_list[0][1]) #2
  print(nested_list[1][-1]) #6
  ```

- we'll have to use nested loops to iterate over these
- List Comprehension in Nested Loop

  ```
   empty_board = [['x' for x in range(1,4)] for n in range(1,4)]
  print(empty_board)
  ```

Dictionaries

- Are used to describe data in more detail
- Data structure that consists key-value pairs

- `keys` are used to describe data and `values` are used to represent data
- `keys` are usually num or strings but `values` can be anything from boolean to nested dictionaries
- There is no fixed order for items in a dictionary, they are unordered

Accessing Individual Values

```
# creating a dictionary
cat = {
  "name": "Kitty",
  "age" : 5,
  "isCute": True
}

# accessing properties
cat["name"] # Kitty
```

Iterating Over Dictionaries

- use `for v in dict.values()` to iterate over values
- use `for k in dict.keys()` to iterate over keys
- use `for k,v in dict.items()` to iterate over both at the same time

Using `In` with Dictionaries

```
cat = {
  "name": "Kitty",
  "age" : 5,
  "isCute": True
}

###### DOES A DICTIONARY HAVE A KEY?
print("name" in cat) # True
print("nails" in cat) # False

###### DOES A DICTIONARY HAVE A VALUE?
print("Kitty" in cat.values()) # True
print("Pussy" in cat.values()) # False

# ! THE BELOW CODE WILL THROW AN ERROR
if cat["nails"]:
  print("nail")

# ! THE CORRECT WAY
if "nails" in cat:
  print("nail")
```

Dictionary Methods

clear()

- empty out the dictionary

  ```
  d = dict(a=1,b=2)
  print(d ) # {'a':1, 'b':2}
  d.clear()
  print(d) # {}
  ```

copy()

- copies one dictionary to another variable but they don't point to the same in memory

```
d = dict(a=1,b=2)
c = d.copy()
print(c) # {'a':1, 'b':2}
print(c == d) # True
print(c is d) # False
```

fromkeys()

- used to dynamically add items to an empty dictionary
- used to set default values

```
# add a fixed 'unknown' value to all the keys for new user
new_user = {}.fromkeys(['name', 'age', 'score'], 'unknown')

print(new_user) # {'name': 'unknown', 'age': 'unknown', 'score': 'unknown'}
```

- if you don't put a list as the first argument and maybe put a string instead then it will iterate over the characters of the strings
- can also use range as an iterable object here

```
new_dict = {}.fromkeys(range(1,10), 'None')
```

get()

- if we have the specified key, we get its value otherwise we get `None`

```
d = dict(a=1,b=2)

# NORMAL WAY
print(d['a']) # 1
print(d['c']) # KeyError

# USING GET METHOD
print(d.get('a')) # 1
print(d.get('c')) # None
```

pop(key)

- removes the specified key and its value
- gives an error if the key doesn't exist
- returns the value of the item that is removed

```
d = dict(a=1,b=2)

print(d) # {'a':1, 'b':2}
print(d.pop('a')) # 1
print(d) # {'b': 2}
```

popitem()

- removes any random key from the the dictionary
- does not take any argument

update()

- add everything from one dictionary to another one

```
a = dict(a=1,b=2)
b = {'c':3}

#  update `b` with `a`
b.update(a)

print(b) # {'c': 3, 'a': 1, 'b': 2}
```

- it doesn't overwrite prev existing values
- it does overwrite duplicate key-values
- it only updates and does not remove anything if we pass a dictionary w fewer properties

Dictionary Comprehension

- iterates over keys by default
- `.items()` can be used to iterate over both keys and values

Conditional Logic with Dictionary Comprehension

Just add if else according to logic man ughh

```
new_list = {num: 'even' if num%2==0 else 'odd' for num in range(1, 10)}

# new_list = {1: 'odd', 2: 'even', 3: 'odd', 4: 'even', 5: 'odd', 6: 'even', 7: 'odd', 8: 'even', 9: 'odd'}
```

Tuples and Sets

Tuple

- Ordered collection of items `(item1, item2)`
- it is immutable and cannot be changed
- we can also have nested tuples

WHY USE THEM?

- they are faster if you don't mind immutability
- makes code safer, value doesn't get changed
- can act as valid keys in dict, unlike lists
- some methods return tuples
- An example for using tuple in a real case can be days of a week

  ```
  days = tuple(monday, tuesday, wednesday, thursday, friday, saturday, sunday)

  # access data with index
  print(days[0]) # monday
  ```

TUPLE METHODS

- `count(item)` - returns how many times the specified item is in tuple
- `index(item)` - returns the index of the specified item's first instance

Sets

- just like mathematical sets, created by `{}`
- cannot have duplicate values
- are unordered, hence can't be access by index

  ```
  # defining sets
  s = {1, 2, 3}

  s2 = set({4, 5, 6})
  ```

WHY USE THEM?

Suppose I have 100 people coming from a lot of cities and we want to find the unique cities. We can create a set out of the list of cities.

```
projects = ['java', 'HTML', 'python', 'java', 'css', 'python']

# finding unique stack
stack = set(projects)
```

SET METHODS

- `add(item)` - adds item to the set; ignores if item is already there
- `remove(item)` - removes item; throws error if item isn't there

- `discard(item)` - removes item; doesn't throw error if item isn't there
- `copy()` - creates a copy of the set
- `clear()` - removes all the items from the set
- There are many mathematical methods with sets like union and intersection

Set Comprehension

- same stuff as we have for list and dict comprehension
- useful when we convert other data types into set to get unique instances

  ```
  s = {x **2 for x in range(5)}
  print(s) # {0, 1, 4, 9, 16}

  print({letter for letter in "hello"}) # {'h', 'o', 'l', 'e'}
  ```

Functions

A function is a block of code that performs similar tasks every time it is called. It can take inputs and returns output when the return keyword is used.

```
# defining a function
def name_of_function():
  # all the code
  print('Hi')

# invoking function by calling it
name_of_function() # Hi
```

THE `RETURN` KEYWORD

- returns/export a value from a function that can be stored in a variable to use it later
- it exits the function right away and anything below it will be ignored
- it pops the function off the call stack
- we can return anything and not just a single item

```
def greet_user():
  return 'Hello Sir!'

# getting returned value
greeting = greet_user()
print(greeting) # Hello Sir!
```

PARAMETER

Functions accept input and perform the tasks based on those inputs. It makes them more dynamic and reusable.

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

print(square(4)) # 16
print(square(6)) # 36
```

- If we create a parameter while defining functions, we must also pass argument when calling it otherwise there will be an error.
- We can have as many parameters as we want by separating them with comma

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

  print(add(4, 8)) # 12
  print(add(6, 6)) # 12
  ```

- It's better to have relevant parameter names

Parameters vs Arguments

Parameters are the variable in a method definition, when a function is defined, whereas Arguments are the data we pass into while calling the function

For example, in the above code block, `a` and `b` are the parameters where as `4` and `8` are the arguments

DEFAULT PARAMETERS

- we can set up a default value for parameters so if we don't provide an argument, the function runs w the default parameter and doesn't throw an error
- if we do provide an argument, the function overwrites the default parameter

```
def exponent(num, power = 2): # default value as 2
  return num**power

print(exponent(2,5)) # 2**5 = 32
print(exponent(4)) # 4**2 = 16
```

- allows you to be more defensive and helps avoid error with incorrect parameters
- default parameters can be anything from a string to list, dictionary, booleans or even another function
- parameters are assigned in order so try to it set it up at the end or make sure each parameter has a default value

KEYWORD ARGUMENTS

- can be used to specify which argument corresponds to which parameter
- hence, the order of the arguments doesn't matter anymore

```
def expo(x, y):
  return x**y

# normal
print(expo(3, 2)) # 9
print(expo(2, 3)) # 8

# keyword arguments
print(expo(y=2, x=3)) # 9
```

When we use = while defining a function, we set up a default parameter whereas when we use = while invoking a function we make a keyword argument

PARAMETER ORDERING

If we're gonna have all of these things in a function, this is the order they should follow

1. parameters
2. *args
3. default parameters
4. **kwargs

In case we have both `*args` and `default parameters` then we would need to pass the values for default parameters as keyword arguments otherwise `*args` will eat up those arguments.

args and kwargs

*args

- special operator we pass to functions
- gathers any remaining arguments after using it as a tuple
- can be called anything - `*some_name`

```
# when we no. of parameters to be passed
def sum_all_nums(a, b, c):
  return a+b+c

print(sum_all_nums(2,5,7)) #14

# when no. of parameters isn't fixed
def sum_nums(*args):
  total=0
  for num in args:
    total += num
  return total

print(sum_nums(2,5,3)) #10
print(sum_nums(1,2,3,4,5)) #15
```

- we can have as many arguments as we want without having corresponding parameters for them

**kwargs

- special operator passed w functions
- looks for keyword arguments
- gathers all the remaining keyword arguments after it is used and turns them into a dictionary
- can be called anything we want - `**some_name`

```
 def fav_colors(**kwargs):
   for k, v in kwargs.items():
     print(f"{k}'s fav color is {v}")


fav_colors(durgesh="red", john="black", tanu="pink")
'''
durgesh's fav color is red
john's fav color is black
tanu's fav color is pink
'''
```

- we can pass as many keyword arguments and it'll print the fav color for each of them

Scope

- where our variables can be accessed
- whenever we define a variable inside a func, it can only be used inside the function

Global

- when we have variables outside a function they are called global
- we can alter a global value from inside a function in the usual way
- we need to use `global` keyword to reference the variable from global scope (here we basically tell function that we want to use global variable cuz by default function looks for a local variable)

  ```
   total = 0

  def inc():
      global total # if we don't have this line, we get an error
      total+=1
      return total

  inc()
  ```

- we don't need to use `global` keyword if we only want to access and not change it

nonlocal

- lets us modify a parent's variables in a child (nested) function

  ```
   def outer():
    count = 0
    def inner():
      nonlocal count # tells function we're not looking for a local variable (works similar to global)
      count += 1
      return count
    return inner()
  ```

- works similar to `global` keyword but used for cases like above where a parent's variable needs to be manipulated but it's not in the global score

Documenting Functions

- **Doc Strings** can be used to explain what a function does
- use `""" msg """` inside a function to write a doc string
- they can even be accessed using `function_name.__doc__`

```
 def say_hello():
   """ A simple function to return a hello msg to user"""
   return "Hello Friend!"

print(say_hello()) # Hello Friend!
print(say_hello.__doc__) # A simple function to return a hello msg to user
```

- we can see the doc string even for the built-in functions `print(print.__doc__)`

Unpacking

TUPLE UNPACKING - using `*` as an argument

- `*collection` - breaks down collection into its single items so each item can act as an argument instead of whole collection being an argument

```
 def print_args(*args):
   print(args)

nums = [1,3,2,4]

print_args(1,3,4,5) #(1, 3, 4, 5)
print_args(nums) #([1, 3, 2, 4],)
print_args(*nums) #(1, 3, 2, 4)
```

So if we have to a pass a long list to a function call where we want to use all the single items from the list, we can use unpacking to break down list items into separate individual arguments

- works the same with tuples too

DICTIONARY UNPACKING - using `**` as an argument

- `**dictionary` - used to unpack dictionary into its individual key-value pairs

Unpacking with Parameters

```
 def display_names(first, second):
   print(first, second)

name = {"first": "John", "second":"Doe"}

display_names(first="Colt", second="Steele")
# Colt Steele -> first = Colt, second = Steele

display_names(name)
# TypeError -> first = names, second = ???

display_names(**name) # UNPACKED
# John Doe -> first = name[first], second = name[second]
```

Unpacking with Kwargs

```
 def add_nums(a,b,c,**kwargs):
   print(a+b+c)
   print('some more code')
   print(kwargs)

dict1 = dict(a=1, b=2, c=5)
dict2 = dict(a=1, b=2, c=5, d=15, name="Ron")

add_nums(**dict1)
'''
8
some more code
{}
'''


add_nums(**dict2, cat="red")
'''
8
some more code
{'d': 15, 'name': 'Ron', 'cat':'red'}
'''
```

Lambda and BuiltIn Functions

Lambdas

- lambdas are single line functions that have no name
- sometimes are called anonymous functions
- only has a single expression w/o the return keyword
- the result can be stored in a variable but not so common use

  ```
   add = lambda a, b: a + b

  print(add(3,10)) # 13
  ```

- they are not used that much

Map

- a standard built-in function
- takes 2 arguments - a function(lambda) and an iterable
- runs the passed function for each iterable and returns a collection called *map object*

```
 nums = [2,4,5,6,8]

doubles = map(lambda n: 2*n, nums)

print(doubles) # <map object at 0x000001B8AF75EDA0>
print(list(doubles)) # [4, 8, 10, 12, 16]
```

- this is one of the common use case of lambdas
- it returns a map object which needs to be converted into list to print it
- however, the map object is iterable

Filter

- it basically filters out values out of a collection based on the lambda
- takes 2 argument, a function (usually lambda) and an iterable collection
- runs lambda for each iterable and returns a collection called *filter object*
- filter object has only the values that return true to the lambda

```
# Filter out the evens from the list
nums = [1, 2, 3, 4, 5]

even_nums = filter(lambda n: n % 2 == 0, nums)

print(even_nums) # <filter object at 0x000001B978C0EDD0>
print(list(even_nums)) # [2, 4]
```

- whenever we use lambda with filter, lambda needs to be a boolean expression (should return True/False)

Things to Remember?

We can use **map** and **filter** together to create some complex functionality.

In most cases, we can achieve the same output as map and filter using list comprehensions and it's perfectly okay to use that. It's just good to know these things exist too.

We might see the lambda way in certain external libraries so it's good to know.

Built-In Functions

all() & any()

all - Returns true if all elements of iterable are truthy (or if the iterable is empty)

```
print(all([0,1,3,4])) # False since 0 is falsy

names = ["John", "Jack", "Jay"]
is_J_first = [name[0]=="J" for name in names]

print(is_J_first) # [True, True, True]
print(all(is_J_first)) # True
```

any - Returns true if any one of the iterable is truthy

```
print(any([0, '', 4, None])) # True, since 4 is truthy
```

> When iterating over lists, we can avoid using list comprehensions when we're using `any` or `all` method and it'll work the same way. Incase we need the results for later use then using list comprehension would make good sense.

sorted()

- Returns a new sorted *list* from the items in iterable
- Works on anything that is iterable

```
numbers = [6, 12, 15, 11]

print(sorted(numbers)) # [6, 11, 12, 15]
print(sorted(numbers, reverse=True)) # [15, 12, 11, 6]
```

- Unlike the `sort` list method, it doesn't actually modify the list

max() and min()

max - Returns the largest item in an iterable or the largest b/w 2 or more passed arguments

min - Returns the smallest item from an iterable or b/w the passed arguments

```
max(3, 99, 67) # 99
min(3, 99, 67) # 3

max('c', 'd', 'a') # 'd'
min('c', 'd', 'a') # 'a'

max("hello world") # 'w'
min("hello world") # ' '
```

reversed()

Reverses any iterator but returns a *reversed object* and not a list

```
 for letter in reversed("hello"):
   print(letter) # 'o' 'l' 'l' 'e' 'o'

# using slice
print("hello"[::-1]) # 'olleh'

# doing above with reversed
rev = "".join(list(reversed("hello")))
print(rev) # 'olleh'
```

- Unlike the reverse list method it does not actually modify the iterator

len()

- Pretty basic, right? It returns the length of an object. List, string, tuple, dictionary (counts the np. of keys), you name it!
- BTS, len uses a dunder (*d*ouble *under*score) method called **__len__()**

  ```
  'hello'.__len__() # 5
  ```

- We are not supposed to directly call this method but using this we can create our methods for our classes

abs(), sum(), & round()

abs - returns the absolute value of a number which can either be an integer or a floating point number

```
 abs(-25) #25
abs(-75/8) #9.375
abs(2.5) #2.5
abs('20') # ERROR
```

sums - takes an iterable with an optional *start* argument and returns the total sum of start and all the items of iterable (left to right)

The default value of the *start* argument is 0.

```
 sum([1, 2, 3]) # (1+2+3) = 6
sum([1, 2, 3], 10) # 10 + (1+2+3) = 16
sum([1, 2, 3], -11) # -11 + (1+2+3) = -5
```

round - returns number with *n digits* after the decimal as specified. If not specified, it rounds off to the nearest integer.

```
 round(10.2) # 10
round(-5.6) # -6
round(75/7) # 11
round(75/7, 3) # 10.714
```

zip

- Zips and returns the given iterators into a single collection called *iterator of tuples*
- In the iterator of tuples, first element will be a tuple of first element of all the iterators passed in the zip function (see code below)

```
 first_zip = zip([1, 2, 3], [4, 5, 6])

first_zip # <zip object at 0x000002257C54B100>

list(first_zip) # [(1,4), (2,5), (3,6)]
dict(first_zip) # {1:4, 2:5, 3:6}
```

- It stops zipping as soon as the shortest iterable runs out of the elements
- **eg** - If n1 has 5 elements & n2 has 8 elements and they are zipped, the result iterator will only have 5 elements
- We are not limited to just use numbers, we can even combine strings and anything else

We can also unpack a list inside a zip function

```
three_by_two = [(0,1), (1,2), (2,3)]

z = list(zip(*three_by_two))
# z = list(zip((0,1), (1,2), (2,3)))
# z = [(0, 1, 2), (1, 2, 3)]
```

In the above code `*` is unpacking the list *three_by_two* and taking out all of the tuples from it to pass them individually inside the zip function

> All of these methods need not be remembered. They are just one google search away.

Debugging and Error Handling

`raise` Your Own Errors

- In Python we can also throw our own errors using the **raise** keyword.
- Helpful when creating your own kinds of exception and error messages

```
raise ValueError('invalid value')
# throws a value error
```

- We have to raise errors in a decent way with errors that make sense.

`try/except` Blocks

- In python we should use try/except block to handle errors and do something about them

```
foobar # NameError (since it makes no sense as it isn't defined)

# using try/except blocks
try:
  foobar
except:
  print("There's a problem") # "There's a problem" (Prints custom msg instead of throwing a NameError)
```

- Even though we can do it, we should not catch every single error because then we would not be able to identify what went wrong with our code.
- We can try to be more specific about the type of error in except block and that way it will only catch a certain type of error

  ```
  try:
    foobar
  except NameError:
    print("There's a problem")
  ```

try, except, else, and finally

try - attempts to do something **except** - runs if there's an error in `try` block **else** - runs if except doesn't run **finally** - runs at the end no matter what

```
try:
  num = int(input("Enter a num")) # runs first
except:
  print("That's not a number") # runs if user enter something other than a number
else:
  print("Nice number, I'll do it") # runs if a number is entered
finally:
  print("Programs ends") # runs at the end anyway
```

- try and except are the ones used the most
- multiple errors can be combined inside a tuple to be handled in a single *except* block
- we can also give a name to error and handle it inside the except block

  ```
  try:
    ....
  except TypeError as err:
    print(err)
  ```

Python Debugger - Debugging with PDB

- Used to handle the errors that are not intentional, i.e, we didn't raise it, we didn't expect it
- Handles errors whether it's a huge shouting error or a tiny unexpected bug that isn't even breaking our code
- `pdb` is a module that we import and we a single method to pause the execution of our code
- We can set breakpoints in our code anywhere we want just by inserting this line

  ```
  import pdb; pdb.set_trace()
  ```

When python encounters this line it pauses. It doesn't quit, it doesn't skip everything else, it just pauses and then we can check things out line by line or in our terminal.

PDB Gotcha

If you have code with variables names that match the *commands of PDB* then there may be unexpected results while accessing them in the terminal.

```
def add_numbers(a,b,c):
  import pdb; pdb.set_trace()

  return a+b+c

add_numbers(1, 2, 3)
```

In the above example, if we have a variable called *c*. When we type *c* in terminal to access it, we will accidentally hit the *continue* command of pdb instead. In such case we use *p* command to access the variable.

```
p c
```

Types of Errors

SyntaxError

- Occurs when python encounters incorrect syntax
- Usually due to typos

```
def first: # missing the parenthesis

None = -1 # `None` can't be a variable
```

NameError

- occurs when a variable isn't defined or hasn't been assigned a value

```
print('name') # name

print(name) # NameError: 'name' isn't defined
```

TypeError

- mismatch of data types
- occurs when an operation or function is applied to wrong data type

```
len(5) # TypeError: 'int' has no len()

"awesome" + [] # TypeError: can't concatenate 'str' and 'list'
```

IndexError

- occurs when we try to access an element in a list using an invalid index

```
list1 = ["hey", "hello"]

list1[4] # IndexError: list index out of range
```

ValueError

- occurs when a built-in operation or function gets an argument that has the right type but inappropriate value

```
int('55') # 55 (as an integer)

int("joe") # ValueError: invalid literal for int()
```

KeyError

- Occurs when a dictionary does not have a specified key

```
d = {}

d["foo"] # KeyError: 'foo'
```

AttributeError

- Occurs when a variable does not have an attribute

```
"awesome".foo # AttributeError: 'str' object has no attribute 'foo'

[1,2,3].hello() # AttributeError: 'list' object has no attribute 'hello'
```

There are a lot of errors and they can referenced through the python docs. Also whenever you get an error, just google it and it'll be fairly simply to find that error.

Modules

Why use Modules?

- keeps python files small
- reuse code across multiple files
- a module is just a python file

Built-In Modules

- comes with python by default
- we import them to our file to use it
- there are a ton of them, refer to docs

```
import random

random.choice(['apple', 'banana', 'melon']) # picks out a random value
```

- we can give the module an alias

```
import random as ran

ran.randint(1, 100) # picks a random no. b/w 1-100
```

- we can also import just parts of a module

```
from random import randint

randint(1, 100) # picks a random no. b/w 1-100
```

> Try to only import the methods you actually need instead of importing the whole module

Custom Modules

- just a python file we created that we can import into other files
- let's say we have a file called `file1.py` with the following code

```
 def fn():
   print('hello friend')


def fn1():
   print('Good Morning Sir')
```

- Now let's say we need to use the above functions in another file `file2.py`

```
 import file1


file1.fn() # prints 'hello friend'
```

- They do need to be in the same directory otherwise we'd need to provide proper paths

__name__ variable

- every python file has __name__ variable
- it's value is the name of the file unless it's the primary file (something like app.py)
- for primary files , it's value is __main__

> when import finds a file it runs the code inside that file which can sometimes result in extra print statements

we can prevent the above issue by using a condition where the code should only execute if name variable is eq to main

External Modules

- modules created by python devs all around the world
- external modules are downloaded using **pip**

```
python3 -m pip install NAME_OF_PACKAGE
```

autopep8

- Formats code based on the autopep style guide

```
autopep8 -i file_name.py
```

- there are more options like `-i` & `-a`

HTTP Requests

- We can make HTTP requests with python too
- There are a bunch of packages to help with that and we have used one called `requests`

Headers

- Meta data that tell what type of data is expected
- We can use them with get requests to specify what kind of data we want from an API (json, html, etc.)

Query Strings

- A way to pass data to server as part of GET request
- Comprises of key value pairs - `https://yodkwtf.com/?key1=value1&key2=value2`
- Additional info passed with the request
- We can have as many query strings as we want

Object Oriented Programming

- Encapsulation & Abstraction
- Classes and instances
- Attaching methods and properties to classes

What is OOP?

- Most programming have OOP, not python specific
- Method of programming that attempts to model some process/thing in the world as a **class** or **object**
- Kinda allows us to create our own type

Classes

- A blueprint for objects
- Can contain methods and attributes (similar to keys in a dict.)
- For eg, a user class is a blueprint for every individual user we create

Object

- Instance of a class (based on the class blueprint)
- Has the methods and properties of the class
- There are so many built-in classes too
- For eg, every list we create is an instance of the class *list* and every list method comes from the list class

Why OOP?

- An easier way to organize and structure our code in a more humanly way
- Everything can be grouped into different classes
- Isn't a compulsion, there are languages that don't even support OOP and can still do all the same things
- Breaking things down and classifying them

Encapsulation

- Grouping of public and private attributes into a class

private attributes

- are the properties that don't need to be exposed (used) outside a class
- we can use them but it's a convention to let other devs know that we shouldn't
- their names starts with an underscore (_cards)

Abstraction

- Exposing only "relevant" (bare min.) data from a class, hiding private attributes and methods (aka inner workings)

Classes & Objects

- we can create empty classes using `pass` keyword
- if 2 instances of a class are same, they still point to different place in memory

Naming Convention for Classes

- Created with the keyword `class`
- Use CamelCase - convention not compulsion
- Should be singular - User not Users

Instance Attributes

- Classes have a special **__init__** method which gets called every time we create an instance of that class
- `init` method takes a parameter called **self** which points to that particular instance of a class we're working with
- Doesn't have to be called self but it's common convention
- This init method creates all the instance attributes

Instance Methods

- we can add functions to classes which are called methods
- these functions can be accessed by all the instances of that class
- we must pass **self** as the first parameter to all the methods even when there's no use of it, python is expecting it or else it'll throw an error
- we can also mutate the pre defined attributes inside these methods

Class Attributes

- Lives on the class itself rather than defined by the instance
- Shared by all the instances
- Only defined once
- Used far less than instance attr.
- Can be defined right above all instance att. w/o needing the **self** keyword
- Can be used as an attribute on the individual instances too

Class Methods

- Not concerned w instances, but the class itself
- Not used that often, very rare
- Prefixed with the **@classmethod** decorator
- Used when we don't need any data about the instances & we're doing something on the class level
- The class itself is passed as an argument (just like *self* but conventionally called **cls**)
- Common use case can be validating, or converting coming data from one form to another before creating a new instance from that data

Repr Method

- Another dunder method `__repr__(self)`
- Allows us to format what we return from an instance into a readable string
- Whatever we return from this func is the default return when we print that instance otherwise we get the instance in an unreadable format
- Can change `<__main__.User object at 0x000001FE7C3BFEE0>` to `john doe`

Underscores, Dunder Methods, etc.

- `__init__` called dunder methods
- have double underscore on both sides
- special methods python looks for
- `__msg` - called name mangling
- python changes the names of these variables behind the scenes
- basically `__msg` inside a `User` class gets renamed to `_User__msg` instead
- makes a attribute particular to a class to avoid name collision during inheritance
- `_name` - a convention to write private attributes
- there's no such thing as an actual pvt attr. in python but it's for other devs to alert them
- there pvt attr. can only be used by other attr. internally inside classes/functions

Inheritance

- When a class inherits properties from another class (base or parent class)
- Done by passing the parent class as na argument when defining child class
- Multiple inheritance is also possible
    - child element can inherit properties from multiple parent classes
    - the order in which parent classes are passed matters
    - not that commonly used

Properties

- Makes it easy to work with methods inside classes as we don't have to call them with arguments
- They can just be accessed like attributes
- Decorators are used above the methods for this purpose
- Kinda like syntactic sugar

Super()

- Used to call the parent class inside the child's init method
- Automatically passes *self* as the first argument
- Helps reduce code duplication as we don't have to call separate init for all the child classes

```
 super().__init__(name, species) ⍰
Animal.__init__(self, name, species) ⍰
```

Method Resolution Order (MRO)

- When a class is created python sets up MRO behind the scenes
- It is the order in which python will look for methods on instances of the class
- A lot of complexity behind the scenes on python decides the order
- MRO can be referenced in 3 ways -
    - **__mro__** dunder attribute on class
    - **mro()** method on class
    - builtin **help(cls)** method -> most readable way
- Not that common topic

Polymorphism

- An object can take on many(poly) forms(morph)
- Two most important practical applications
    - same class method works similar way for different classes eg. a speak method on class Dog, Cat, etc.
    - same operation working for different kinds of objects eg. len('string'), len([1,2,3])

Special Methods

- Called magic methods, tell python how to handle certain operations behind the scenes
- All of them are dunder methods

- Can be overwritten inside classes
- Can allow us to mutate how certain operations work in python

Iterators and Generators

Iterator

- An object that can be iterated upon, something we can run a for loop over
- Behind the scenes, for loop calls the **next()** method and it returns the next item in the loop
  - One item at a time
  - Until it raises a *StopIteration* error, i.e, at last item

Iterable

- An object that returns an Iterator when iter() is called
- When we loop over something (str, list, etc.), for loop calls the iter() function on it behind the scenes and makes it into an iterator

Generators

- Subset of iterators
- Every generator is an iterator but not vice-versa
- Easy way to create iterators
  - One way is to use *generator functions* using **yield** keyword
  - Another way to create generators is w *generator expressions*

Generator Functions

- Just like a normal function but uses *yield* instead of *return*
- Can yield multiple times unlike normal function that return only once
- Returns a generator object when invoked
- It only stores one thing at a time until it's changed any time we call the next method on it

Generator Expressions

- Easier way to create generators
- Look like list comprehensions for lists
- We use **()** instead of **[]**
- It's more like Tuple Comprehension in syntax

```
 g = (num for num in range(1, 10))
# g here is generator object created from generator expression
```

Decorators

Higher Order Functions

- We can pass functions as arguments inside other functions
- We can also nest functions inside other functions
- We can also return functions from other functions
- Inner functions can access outer functions scope (variables) aka **closure**

> Colt Steele's "Python Course - Higher Order Functions" video has a great example of closure

Decorators

- Decorators are functions that wrap other functions enhancing their behaviors
- They then return the wrapper function, not the func that has been wrapped
- They are examples of Higher Order Functions
- They have their own special syntax `using @` (syntactic sugar), though it's not mandatory to use them

Decorators with Different Signatures

What to do if the wrapped functions take different number of arguments?

In such cases we pass **args** and **kwargs** as the function parameters for the wrapper function so it can take none to any number of arguments.

```
# BOILER PLATE FOR A DECORATOR FUNCTION

from functools import wraps

def main(fn):
  @wraps(fn)
  def wrapper(*args, **kwargs):
    # some tasks
    result = fn(*args, **kwargs)
    # some tasks
    return result
  return wrapper

@main
def function_name():
  return 'something'

function_name()
```

Decorators Functions Taking Arguments

- We can also create decorator functions that take arguments
- We have to add an extra layer of function which will take the argument passed and return a new wrapper function to wrap our fn to be wrapped

```
from functools import wraps

def main(val):

  def inner(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
      # do something with `val` maybe
      result = fn(*args, **kwargs)
      # some tasks
      return result
    return wrapper
  return inner

@main(some_val)
def function_name():
  return 'something'

function_name()
```

Testing

Done to test whether block of code is running well or throwing errors

Why Test?

- Reduce bugs in code
- Ensures that your bugs stay fixed
- Ensure new features don't break old ones

Writing tests can slow you down. It's not necessary to have them for all your projects - big or small. It's more up to personal preference.

It's a good idea to write them you have a large scale project with customers. clients, multiple developers, etc. depending and working on it.

Test Driven Development (TDD)

- Tests are written first
- Then write code to make those tests pass
- Once tests pass, the feature is complete
- Just one of the way to write tests, you can change the order as per your preference

"Red, Green, Refactor" Way

It's a just mantra that people follow while doing TDD

1. Red - Write a test that fails
2. Green - Write the minimal amount of code to pass test
3. Refactor - Clean up code, while ensuring tests still pass

Assertions

- `assert` keyword accepts an expression to check if it's true
- Raises an *AssertionError* if expression is falsy
- Returns *None* if expression is truthy
- Accepts option error msg as second argument

```
 def add_positive_nums(x,y):
   assert x > 0 and y > 0, "Both the numbers must be positive"
   return x+y

print(add_positive_nums(2, 5)) # 7
add_positive_nums(3, -5) # AssertionError: Both the numbers must be positive
```

If a Python file is run with the **-O** flag assertions will not be evaluated.

Therefore don't make your code dependent on assertions as they can be ignored.

doctests

- allows us to write tests inside doc strings
- we have to write code in a certain way like inside of a REPL
- if we write tests this way, it can also work as a documentation for other devs along with being a test
- there's a special command to run the doctests w/o having to evaluate anything

```
python -m doctest -v file_name.py
```

This will show all the tests that ran, expected output, actual output, no. of passed and failed tests.

- We have to be really careful while mentioning expected output inside doctests. Things like quotation marks, empty spaces, list (or dict) order, etc. can really be a pain at times.

Unit Tests

Unit testing is a technique where we test small individual components (units) as we go

unittest

- Module used for this that comes with a lot of built in assertions
- Unit tests can be written as classes that inherit from `unittest.TestCase`
- This gives us a lot of assertions
- Tests are run by calling `unittest.main()`

There are a bunch of different assertions method provided by the module. We can refer to the documentation for more info about them.

setUp & *tearDown*

- **setUp** - A function used to run something before every test
- **tearDown** - A function used to run something after every test
- Names are really imp and should be the same

File IO

- Really useful for data science - csv files, datasets, etc.
- Useful for web dev - reading html/css/js files and sending contents

Reading Files

- We can read a file by using the `open(file_name)` function
- `open` returns a file object to us - there's some metadata about the file
- Then we have to read it using the `read()` method
- `open` has a bunch of optional parameters but the file name is a compulsory
- If file can't be opened, an **OSError** is raised
- We can ref. either absolute or relative path

There are a bunch of options which can are there on the python documentation

```
 # open a file object
f = open(file_name.txt)

# print file object (instance of TextIOWrapper)
print(f)

# print the contents of the file
print(f.read())
```

Cursor Movement

When you use a read method on a file for the second time, you'll see an empty string. This is because python reads it using the cursor movement. So once you read the whole file, the cursor moves at the very end. When you again try to read the file, there's nothing left after the cursor from the last time.

seek

used to send the python cursor to a specific place in the file

```
 file = open(file_name.txt)

file.seek(0) # will send it back to the start of the file

file.seek(2) # will send the cursor to the index 2
```

readline

- Used to only read a line
- Useful when reading a large file line by line

```
 file = open(file_name.txt)

file.readline() # read the first line (till `.`)

file.readline() # read the next line
```

readlines

- Reads all lines one by one and puts them in a list

```
 file = open(file_name.txt)

file.readlines() # returns a list of all the different lines
```

Closing Files

When we're done using a file we should close a file so we don't waste resources.

`file.closed` - returns if the file is closed or not (True/False) `file.close()` - closes the file

The `with` statement

Another way to open and read python files but using this statement automatically closes the files so we don't have to manually close them.

```
 with open(file_name.txt) as file:
  file.read()

file.closed # will always return True, we don't have to do it
```

Behind the scenes, `with` uses the `__enter__` and `__exit__` method to initiate and close a file every time it is called no matter what.

Writing Files

- We still have to use `open` function to write to a file
- Have to specify `-w` flag to specify that we're writing

```
with open('file_name.txt', 'w') as file:
  file.write("I am writing this...Wow")
```

- The `write` method doesn't preserve, it rewrites everything
- If the file doesn't exist, python will create a new one with the specified name

File Modes

There are a bunch of different flags (called modes) that we can pass along with the `open` function. They can be seen on the documentation.

- `r` - read a file (default)
- `w` - write to a file (overwriting)
- `a` - append to a file (no overwriting)
- `r+` - read & write to a file (writing based on cursor)
    - common to use with files with pre-existing data
    - only woks with pre-existing files won't create one for us
    - throws error if file isn't found

CSV and Pickling

CSV

- Comma Separated Values
- Common for tabular data

We use specific csv module to read/write csv files in a decent way

Reading CSV Files

2 Different Ways

- **reader** - iterates over rows of CSV as lists
- **DictReader** - iterates over rows of CSV as OrderedDicts

reader is an iterator

Note: We can read data from csv files even if they are not separated by commas, as long as they are separated by the same character. That character is knows as *Delimiter*.

Writing CSV Files

- **writer** - creates a writer object for writing to CSV
- **writerow** - method on a writer to write a row to the CSV

Pickling

We can pickle some data by putting it into a specific pickle file using the pickle module.

Python serialize the data and converts into byte streams and later when we need the data we can unpickle it and convert it back to what it was.

Web Scraping

Web Scraping means programmatically grabbing data from a web page

Three steps: Download, extract data by parsing the html received, use that data

Why Scrape?

If a website doesn't provide an APi to work with their data, we can use scraping to get some data off of their website

Is it ethical?

- Some websites don't want you to do it, doesn't mean you can't do it.
- Best practice: consult the robots.txt file to see what files are allowed to scraped
- It's not illegal or something, more like they are requesting you not to do it
- If you make too many req, try to time them out
- If you keep on doing it your IP can be blocked

Beautiful Soup

- Lets us navigate through HTML with python
- Doesn't download HTML - we need requests module for that

```
from bs4 import BeautifulSoup

html_data = """
some html code returned from the requests module
"""

soup = BeautifulSoup(html_data, "html.parser")
# this will convert the html string into html

# Now we can navigate around the html
print(soup.body) # prints the entire body from html
```

- We have many methods to select specific tags and selectors like *select* , *find*, *find_all*

The best way to know about these methods is to read the official docs

Regular Expressions

There are a ton of regex symbols. You can test them out in any regex editor.

They can vary a little bit based on what language you're working with.

Rules

Characters

- `.` can match everything
- `\` is used to escape a string
- `\.` will only match a .
- `ab` will only match *ab*
- `\d` matches digit *0-9*
- `\w` matches letter, digit, or underscore

There are a bunch of cheatsheets online to refer to these rules.

The capital version of these rules reverse the rule

- `\W` gives anything that is not letter, digit or underscore
- `\D` matches anything that is not a number form 0-9

Quantifiers

They specify how many times something should occur in the text

- `+` matches one or more characters
- `{x}` matches exactly x times
- `{3,5}` 3 to 5 times
- `{4,}` 4 or more times
- `*` zero or more times
- `?` once or none (optional)

Character Classes

These are group or sets of different characters

- `[aeiou]` matches any one of the 5; if we just do `aeiou` then it'll match the exact string with these 5 letters
- `[aeiou]{2}` finds characters whenever any 2 vowels are together
- `[A-Z]` refers to all uppercase letters
- `[A-Z]{4,}` matches all words with 4 or more *uppercased* letters
- `[^k]` matches everything that isn't *k*

Anchors or Boundaries

- `^` start of string or line
- `$` end of string or line
- `\b` word boundary

Logical Operator `|`

Allows us to write optionals strings, for example Mr. | Ms. | Mrs.

- The pipe character selects b/w the strings before and after it
- Sometimes they need to be wrapped inside parenthesis to avoid mis-matches

Regex in Python

```
# import re module
import re

# define a basic phone number regex
pattern = re.compile(r'\d{3} \d{3}-\d{4}')
# here `r` is used to indicate that the string is a raw string so we don't need to escape the backslashes

# search a string with our regex
result = pattern.search('My number is 415 555-4242.')

result2 = pattern.findall('My number is 415 555-4242 and 888 000-3333.')

print(result.group()) # 415 555-4242
print(result2) # ['415 555-4242', '888 000-3333']
```

Conclusion

Thank you for viewing this cheatsheet!

If you found it helpful please check out more of my work on yodkwtf.com or follow me on twitter. I also run a small youtube channel called Yodkwtf Academy.