[Cheat Sheet] - Python Core

Easier sheets: https://pythonprinciples.com/reference/

Fun Facts

- **Python** is a high-level programming language, with applications in numerous areas, including web programming, scripting, scientific computing, and artificial intelligence!
- Variable names: only characters that are allowed are **letters**, **numbers**, and **underscores**. Also, they can't start with numbers.
- Avoid Writing Everything Twice (WET)

Data Types

- **Integers -** Whole numbers
- **Floats -** Decimal numbers
- **Doubles -** Double precision numbers (Better version of floats)
- **Booleans -** True or False; Boolean logic (and/or/not)
- **Strings -** Words/array of characters
- **Lists** - a set of elements that can change
- **Tuples** - a set of elements that cannot change
- **Dictionaries** - data structures used to map arbitrary keys to values.

Constants

- Constants are usaully declared in a module

```
#Constant.py
PI = 3.14
GRAVITY = 9.8
---------------------------------------
#Main.py
import Constant
print(Constant.PI) --> Output: 3.14
```

Operations

- + add
- - subtract
- * multiply
- / divide; creates a FLOAT, not an integer
- ** exponential
- // quotient (minimum divisible (7//2 = 3))
- % remainder of a division (7%2 = 1)
- = assign value
- == check if equal
- != Not equal
- > greater then
- < smaller then
- >= greater or equal to
- <= smaller or equal to

In-place Operators

- += (same as "x = x + ...")
- -= (same as "x = x - ...")
- *= (same as "x = x * ...")

- /= (same as "x = x / ...")
- %= (same as "x = x % ...")
- :=(Walrus operator)
  - assign values to variables within an expression, including variables that do not exist yet.

Strings

- Words/array of characters

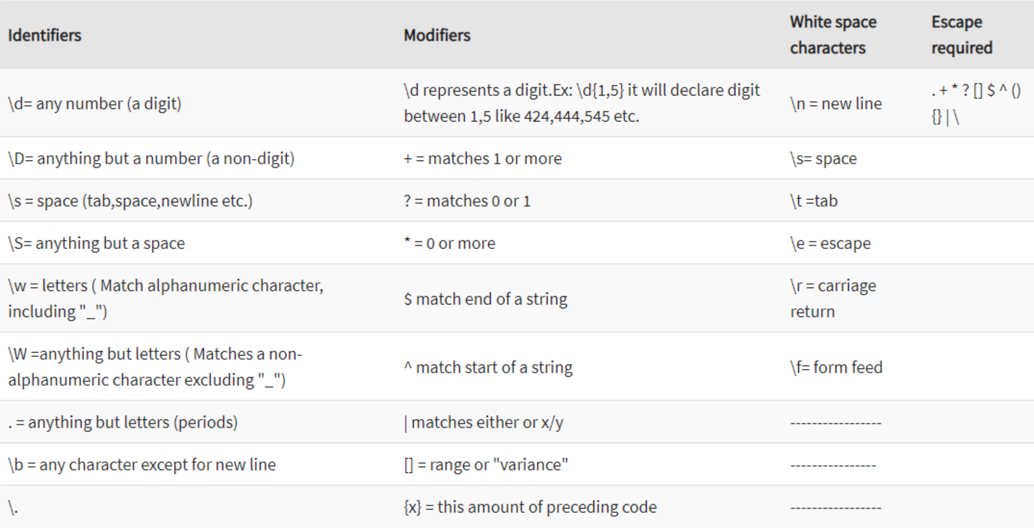
String Formatting

- Similar to placeholders in C
- Negative index makes the array "backwards"
  - onion[-1] = "n"

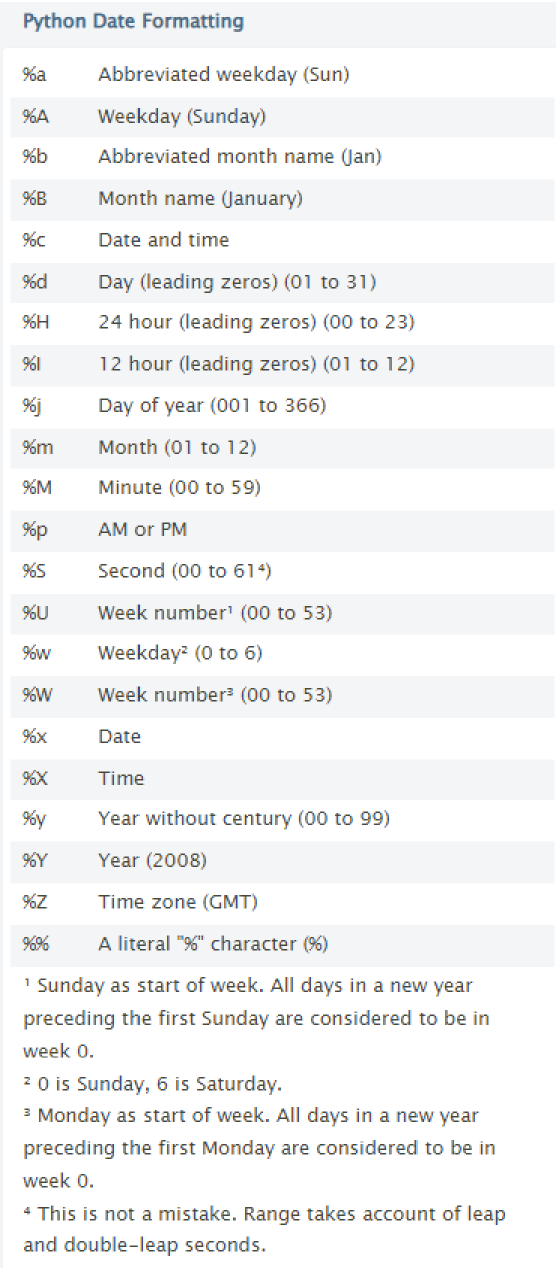
```
#Eg1
nums = [4,5,6]
msg  = "Numbers: {0} {1} {2}".format(nums[0], nums[1], nums[2]) #starts from 0
print(msg)

#Eg2
a = "{x}{y}".format(x=5, y=4)
print(a)

#Eg3 --> Money
print("${:,.2f}".format(123.456))

return [char for char in word] -->
```

Special Characters:

- \ - makes the next special character printable(')



Identifiers	Modifiers	White space characters	Escape required
\d= any number (a digit)	\d represents a digit.Ex: \d{1,5} it will declare digit between 1,5 like 424,444,545 etc.	\n = new line	. + * ? [] $ ^ () {} | \
\D= anything but a number (a non-digit)	+ = matches 1 or more	\s= space	
\s = space (tab,space,newline etc.)	? = matches 0 or 1	\t =tab	
\S= anything but a space	* = 0 or more	\e = escape	
\w = letters (Match alphanumeric character, including "_")	$ match end of a string	\r = carriage return	
\W =anything but letters (Matches a non-alphanumeric character excluding "_")	^ match start of a string	\f= form feed	
. = anything but letters (periods)	| matches either or x/y	-----------------	
\b = any character except for new line	[] = range or "variance"	----------------	
\.	{x} = this amount of preceding code	-----------------	



Python Date Formatting

%a	Abbreviated weekday (Sun)
%A	Weekday (Sunday)
%b	Abbreviated month name (Jan)
%B	Month name (January)
%c	Date and time
%d	Day (leading zeros) (01 to 31)
%H	24 hour (leading zeros) (00 to 23)
%I	12 hour (leading zeros) (01 to 12)
%j	Day of year (001 to 366)
%m	Month (01 to 12)
%M	Minute (00 to 59)
%p	AM or PM
%S	Second (00 to 61[4])
%U	Week number[1] (00 to 53)
%w	Weekday[2] (0 to 6)
%W	Week number[3] (00 to 53)
%x	Date
%X	Time
%y	Year without century (00 to 99)
%Y	Year (2008)
%Z	Time zone (GMT)
%%	A literal "%" character (%)

[1] Sunday as start of week. All days in a new year preceding the first Sunday are considered to be in week 0.

[2] 0 is Sunday, 6 is Saturday.

[3] Monday as start of week. All days in a new year preceding the first Monday are considered to be in week 0.

[4] This is not a mistake. Range takes account of leap and double-leap seconds.

String Functions

- join - joins a list of strings with another string as a separator.
- replace - replaces one substring in a string with another.

- startswith and endswith - determine if there is a substring at the start and end of a string, respectively.
- lower or upper - changes the case of a string
- split or join - turning a string with a certain separator into a list.
- .find(str)
- .count(str) - returns how many times the str substring appears in the given string.
- in/not in - used to check if a string is part of another string

```
print(", ".join(["spam", "eggs", "ham"]))
#prints "spam, eggs, ham"

print("Hello ME".replace("ME", "world"))
#prints "Hello world"

print("This is a sentence.".startswith("This"))
# prints "True"

print("This is a sentence.".endswith("sentence."))
# prints "True"

print("This is a sentence.".upper())
# prints "THIS IS A SENTENCE."

print("AN ALL CAPS SENTENCE".lower())
#prints "an all caps sentence"

print("spam, eggs, ham".split(", "))
#prints "['spam', 'eggs', 'ham']"
```

Metacharacters

- what make regular expressions more powerful than normal string methods
- metacharacter we will look at is **.** (dot)
    - This matches **any character**, other than a new line.

```
import re
string = r"gr.y"
```

```
if re.match(string, "grey"):
    print("OK")
if re.match(string, "gray"):
    print("OK")
if re.match(string, "blue"):
    print("NOT OK")
```

- The next two metacharacters are **^** and **$**.
    - These match the **start** and **end** of a string, respectively.

```
import re
pattern = r"^gr.y$"
if re.match(pattern, "grey"):
    print("Match 1")
if re.match(pattern, "gray"):
    print("Match 2")
if re.match(pattern, "stingray"):
    print("Match 3")

"""
The pattern "^gr.y$" means that the string should start with gr, then follow
with any character, except a newline, and end with y.
"""
```

Regular Expressions

- String manipulation
- Two main tasks:
    - verifying that strings match a **pattern** (for instance, that a string has the format of an email address)
    - performing substitutions in a string (such as changing all American spellings to British ones).

```
import re
pattern = r"spam" #r makes the string "raw"
if re.match(pattern, "spamspamspam"):
    print("Match")
else:
    print("No match")
```

- Functions(if found, returns the match, otherwise returns **None**):
  - re.match function can be used to determine whether it matches at the beginning of a string.
  - re.search finds a match of a pattern anywhere in the string.
  - re.findall returns a list of all substrings that match a pattern.
- These methods include
  - .group() which returns the string matched
  - .start() and .end() which return the start and ending positions of the first match
  - .span() which returns the start and end positions of the first match as a tuple.
- re.**sub**(pattern, repl, string, count=0)

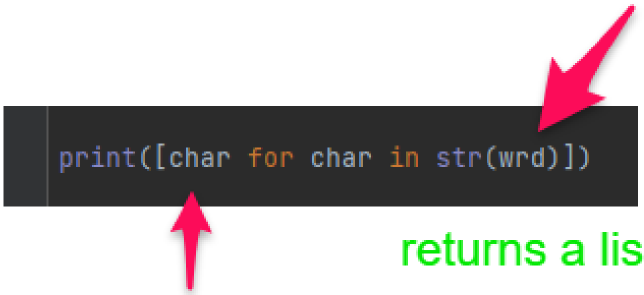
Print Statement

- At the end of a print function, there is a "\n"

```
print("Hello World!") #Prints a string
print(4 + 3) #prints the output as an integer

print(""" A
B
C
""") #creates multple lines (3 in this case)

print("Hello" + "World!") #concatenation --> ads two strings.
print("Hey" * 3) #prints "Hey" 3x

print("Name: {0} \t Age: {1}".format("Gabrie", 20))
#prints(Name: Gabriel     Age: 20)
```

- Spliting numbers/words

input to be separated
must be str()



```
print([char for char in str(wrd)])
```

returns a list of characters

what do you want
the separation to be

Input

- The **input** function (input()) prompts the user for input, and returns what they enter as a string (with the contents automatically escaped).

```
x = input() #assigns an input to x
y = input("Insert age here: ") #prints the statement before taking user's
input
age = int(input()) #transforms the user's input string into an integer
age2 = str(age) #converts the integer "age" into a string
```

If Statement

- You can use **if** statements to run code if a certain condition holds.
- If an expression evaluates to **True**, some statements are carried out.

```
if expression:
    statements

elif expression:
    statement
```

```
else:
    statement
```

Lists

- Used to store items
- [0] is the first item of the list
- Typically, a list will contain items of a single item type, but it is also possible to include several different types.
- Lists can also be nested within other lists.

```
list = ["One", "two", "Three"] #list of strings
list2 = [] #empty list

print(list[0]) #prints the first element of the list
print(list)#prints the whole list

matrix = [
[1,2,3]
[4,5,6]
] #A matrix, being [0][0] the first item (row one colum 1)

ls[0] = 14 --> changes the value in position '0' to '14'
ls.remove(51)
ls.insert(2, 'hello')
print(31 in ls) --> True or False
ls.clear()
```

List Operations

- The item at a certain index in a list can be reassigned.
  - list[0] = x
- Lists can be added

```
list = [1,2,3]
list2 = [4,5,6]
```

```
list3 = list + list2
```

- list items can be checked, with a leaving a True or False output
  - Same as the opposite, using *not in*

```
words = ["spam", "egg", "spam", "sausage"]
print("egg" in words) # prints True
print("tomato" in words) #False
print("tomato" not in words) #True
```

- To convert the result into a list, we used **list** explicitly.

```
x = list(map(function, iteration))
print(x) #prints as a list
```

List Functions

- .append() - adds an item to the end of an existing list.
- len(list) - returns the length of a list
- .insert(position, value) - insert a new item at any position in the list
- .index(value) - finds the first occurrence of a list item and returns its index.
- max(list) - Returns the list item with the maximum value
- min(list) - Returns the list item with minimum value
- .count(item) - Returns a count of how many times an item occurs in a list
- .add() - add new items to the set
- .remove(item) - Removes an object from a list
- .reverse() - Reverses items in a list.
- all(item) and any(item) - take a list as an argument, and return True if all or any (respectively) of their arguments evaluate
  to True (and False otherwise).
- .enumerate() - used to iterate through the values and indices of a list simultaneously
- in/not in - used to check if a item is part of the list
- .pop(index) - removes the item at the given index.

- .reverse() - reverses items in the list.
- .sort() - By default, the list is sorted ascending. You can specify reverse=True as the parameter, to sort descending.
- .reverse() - reverse the list

List Slices

- returns a list of the elements in the index
- Like the arguments to **range**, the first index provided in a slice is included in the result, but the second isn't.
- If the first number in a slice is omitted, it is taken to be the start of the list.
- If the second number is omitted, it is taken to be the end.
- List slices can also have a third number, representing the step, to include only alternate values in the slice.

```
list = [0,1,2,3,4,5,6,7,8,9]
print(list[0:4]) # output = [0,1,2,3]
print(list[:5]) #output = [0,1,2,3,4]
print(list[2:]) #output = [2,3,4,5,6,7,8,9]
print(list[::2]) #output = [0,2,4,6,8]
print(list[::-1]) #output = [9,8,7,6,5,4,3,2,1,0]
```

List Comprehensions

- useful way of **quickly creating lists** whose contents obey a **simple rule**.
- can also contain an **if** statement to enforce a condition

```
cubes = [i**3 for i in range(10)]
evens = [i**2 for i in range(10) if i**2 % 2 == 0]
```

- Trying to create a list in a very extensive range will result in a **MemoryError**.
    - This issue is solved by **generators**

Sets

- **Sets** are collections of unordered items that are **unique**.
- **faster** to check whether an item is part of a set using the **in** operator, rather than part of a list.
- cannot contain duplicate elements.
- Duplicate elements will automatically get removed from the set.
- created with { }

Set Operators:

- The **union** operator **|** combines two sets to form a new one containing items in either.
- The **intersection** operator **&** gets items only in both.
- The **difference** operator **-** gets items in the first set but not in the second.
- The **symmetric difference** operator **^** gets items in either set, but not both.

```
first = {1, 2, 3, 4, 5, 6}
second = {4, 5, 6, 7, 8, 9}

print(first | second)
print(first & second)
print(first - second)
print(second - first)
print(first ^ second)
```

Dictionaries

- data structures used to map arbitrary keys to values.
- Lists can be thought of as dictionaries with integer keys within a certain range.

- **Keys:Values**

```
dictionarie = {"Dave": 1, "Elen":2, "Mary":3}
print(dictionarie["Dave"]) #output = 1

dictionarie2 = {
    "green":[1,2,3,4,5],
    "red":[6,7,8,9],
    "blue":[10,11,12]
} #if key does not exist, it returns a KeyError

dictionarie3 = {
    1:"one",
    2:"two",
    3:"three"
}
print(dictionarie3.get("one")) #output 1
print(dictionarie3.keys()) --> [1,2,3]
print(dictionarie3.values()) --> one, two, three
print(dictionarie3.items()) --> "one":1, "two":2, "three":3
print(fruits['banana']) --> 0.99
```

- A useful dictionary method is **get**. It does the same thing as indexing
  - if the key is not found in the dictionary it returns another specified value instead ('None', by default).

Functions

.get() - same thing as indexing, but if the key is not found in the dictionary it returns another specified value instead.

.keys()

.values()

.items()

... In ...

- "Contem"
- True Or False

```
nums = {
"Dave":24,
}
```

```
print("Dave" in nums)
print(24 in nums)
```

Tuples

- Similar to lists, but **immutable**
  - tuple[0] = "Hey" causes an TypeError
- Tuples are faster then lists, but they are immutable
- created using **parentheses**
- can be used as keys for dictionaries

```
tuple1 = ("Hello", "World!")
tuple2 = "Hello", "World" #same thing as tuple1
empty_tuple = ()
```

1)**Tuple unpacking** allows you to assign each item in a collection to a variable.
2)A variable that is prefaced with an asterisk (*) takes all values from the collection that are left over from the other variables.

```
#1
numb = (1,2,3)
a,b,c = numb
print(a)
print(b)
print(c)

#2
a, b, *c, d = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a)
print(b)
print(c)
print(d)
```

Loops

- While
  - number of iterations is not known and depends on some calculations and conditions in the code block of the loop.
  - while the condition is True, the loop repeats
  - **Break** interrupts the loop
  - **Continue** jumps back to the top of the loop, rather than stopping it. Basically, the **continue** statement stops the current iteration and continues with the next one.

```
while condition:
    code
```

- For
  - used to iterate over a given sequence, such as lists or strings
  - when the number of iterations is fixed
  - Often combined with range
- In
  - "Contem"
  - Eg: the string "hospital" has "hos".
    - if "hos" **in** "hospital" --> True

```
for i in range(10): #iterates 10x
    code
```

Range

- returns a sequence of numbers.
- By default, it starts from 0, increments by 1 and stops **before the specified number**

- If it is called with two arguments, it produces values from the first to the second.
- can have a **third** argument, which determines the interval of the sequence produced, also called the **step**

```
range(10) #Creates a sequence of 10 numbers, from 0 (default)
range(1,10) #Creates a sequence from 1 to 10
range(1,10,2) #Creates a sequence from 1 to 10, 2 by 2
```

Functions

- Makes a sequence of code reusable
- Certain functions, return a value that can be used later.
  - Once you return a value from a function, it immediately stops being executed.

```
def function(arguments):
    code

function(argument)
```

Map

- has to be converted into a list to be printed
- takes a function and an iterable as arguments, and returns a new iterable with the function applied to each argument.
- must be transformed into a list to be printed

```
def add_five(x):
    return x + 5
nums = [11,22,33,44,55]
result = list(map(add_five, nums))
print(result) --> [16, 27, 38, 49, 60]
```

Filter

- similar to map, but filters an iterable by removing items that don't match a predicate (a function that returns a Boolean)

```
ages = [5, 12, 17, 18, 24, 32]
def myFunc(x):
  if x < 18:
    return False
  else:
    return True
adults = filter(myFunc, ages)
for x in adults:
  print(x)
```

Recursion

- functions calling themselves. It is used to solve problems that can be broken up into easier sub-problems of the same type
- can also be indirect(function 1 calls function 2, while function 2 calls function 1)
- The **base case** acts as the exit condition of the recursion.

```
#direct recursion
def factorial(x):
    if x == 1:
        return 1 #this acts as a base case(when a recursion ends)
    else:
        return x*factorial(x-1)

#indirect recursion
def is_even(x):
    if x%2==0:
        return True
    else:
        return is_odd(x-1)

def is_odd(x):
    return not is_even(x)
```

Lambda

- is an anonymous function

```
lambda_func(lambda x: code)
```

Decorators

- provide a way to modify functions using other functions.
    - extend the functionality of functions that you don't want to modify

```
def decor(func):
    def wrap():
        print("============")
        func()
        print("============")
    return wrap

def print_text():
    print("Hello world!")
decorated = decor(print_text)
decorated()

#the code ABOVE has the same output as the code BELOW

def decor(func):
    def wrap():
        print("========")
        func()
        print("========")
    return wrap

@decor --> decorator
def print_text():
    print("Hello World!")

print_text()
```

Numeric Functions

- **abs()** - find the distance between the a number and zero
- **round()** - round a number

- **sum()**- find the sum

Comments

- **#** Single line comments
- **"""________"""**
  - Docstrings
  - Multiple line comments

Exceptions

- They occur when something goes wrong, due to incorrect code or input.
- When an exception occurs, the program immediately stops.
- Main Types:
  - **ImportError**: an import fails;
  - **IndexError**: a list is indexed with an out-of-range number;
  - **NameError**: an unknown variable is used;
  - **SyntaxError**: the code can't be parsed properly;
  - **TypeError**: a function is called on a value of an inappropriate type;
  - **ValueError**: a function is called on a value of the correct type, but with an inappropriate value.
- Other libraries also have their exceptions built-in

Exception Handling

- **Raise**
  - You can raise an error on purpose
  - need to specify the error type

    - In **except** blocks, the **raise** statement can be used without arguments to re-raise whatever exception occurred.
- **Try**
    - contains code that might throw an exception.
    - If that exception occurs, the code in the **try** block stops being executed, and the code in the **except** block is run
- E**xcept**
    - An **except** statement without any exception specified will catch all errors.
    - except blocks can handle multiple errors
- **Finally**
    - To ensure some code runs no matter what errors occur, you can use a **finally** statement
    - "default in C"

```
try:
    code
except(NameError):
    code
exept(ValueError, SyntaxError):
    code
except:
    code

finally:
    code
```

Assert

- "Sanity check"
- An expression is tested, and if the result comes up false, an exception is raised.

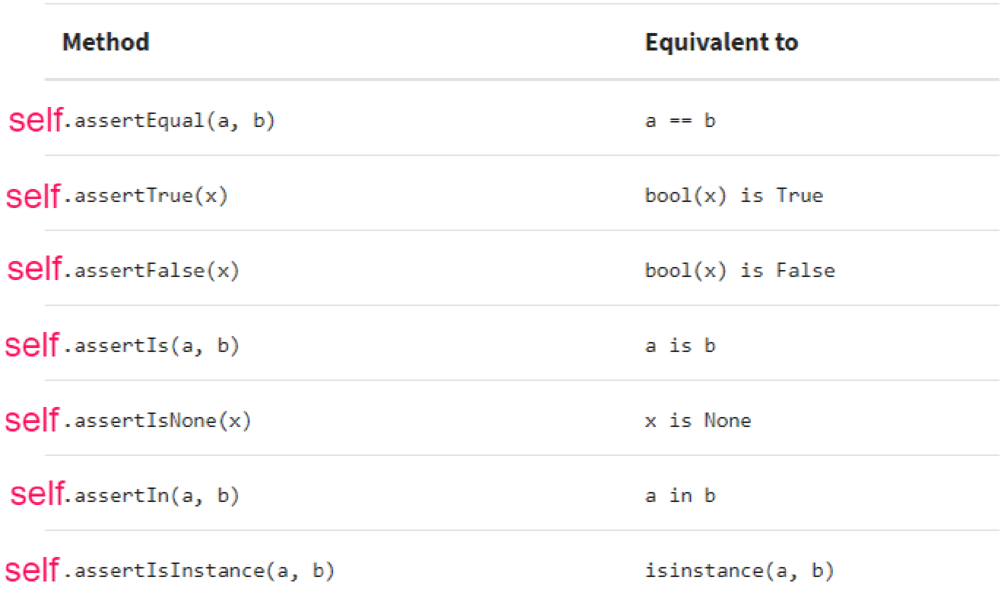
```
assert expression
assert (1+1 == 3), "This is impossible"

print(1)
```

```
assert 1+1 == 2#True statement
print(2)
assert 1+1 == 3 #Statement is false, raising an error and ending the code
print(3)
```

Unit Testing

- https://realpython.com/python-testing/
- https://docs.python.org/3/library/unittest.html
- One of the most usefull tips in programming that will reduce time spent debugging

```
import unittest --> Library

class TestSum(unittest.TestCase): --> Needs to be in a class
    def test_sum(self):
        self.assertEqual(sum([1, 2, 3]), 6, "Should be 6")

    def test_sum_tuple(self):
        self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")

if __name__ == '__main__':
    unittest.main()
    //If on Jupyter unittest.main(argv=['first-arg-is-ignored'], exit=False)
```



Method	Equivalent to
`self.assertEqual(a, b)`	`a == b`
`self.assertTrue(x)`	`bool(x) is True`
`self.assertFalse(x)`	`bool(x) is False`
`self.assertIs(a, b)`	`a is b`
`self.assertIsNone(x)`	`x is None`
`self.assertIn(a, b)`	`a in b`
`self.assertIsInstance(a, b)`	`isinstance(a, b)`

Files

- Important Notes
  - Always use try and finally to ensure the file is closed, even if an error occurs
- Open a file

```
#open in write mode
myfile = open("path", "w")

#open in read mode
myfile = open("path", "r")
myfile = open("path")

#open in append mode
myfile = open("path", "wb")

#open in read and write
myfile = open("path", "r+")
```

```
#open in binary
myfile = open("path", "rb")
```

- Close a file

```
myfile.close()
```

- Reading files
  - After all contents in a file have been read, any attempts to read further from that file will return an empty string, because you are trying to read from the end of the file.

```
file.read() #This will print all of the contents of the file "filename.txt".
file.read(16) #Reads the first 16 bytes of the file

file.readines() #Returns a list on which each element is a line
```

- Writing Files
  - Writes a string in the file
  - If the file does not exists, the file will be created if open in write mode
  - The previous file content is deleted

```
file.write("string") #writes a string in the file
```

Object Oriented Programming(OOP)

Classes

- *"The **class** describes what the object will be, but is separate from the object itself."*
- a class is an object's blueprint
- Classes can have **methods**, which are functions within a class

```
class Cat:
    def __init__(self, name,color, legs):#__init__ is a constructor METHOD
        self.name = name
        self.color = color
        self.legs = legs
    def meow(self):
        print("Meow: "+ self.name)

felix = Cat("Felix", "orange", 4)
print(felix.color)
print(felix.meow())
```

Inheritance

- provides a way to share functionality between classes
- This similarity can be expressed by making them all inherit from a **superclass**, which contains the shared functionality.
  - A class that inherits from another class is called a **subclass**.
  - A class that is inherited from is called a **superclass**.
- The function **super** is a useful inheritance-related function that refers to the
  - **super().method()** calls the **method** of the superclass.

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

    def __str__(self):
        return "this is how I want to print an object: " + self.name
class Cat(Animal):
    def purr(self):
        print("Purr...")

class Dog(Animal):
    def bark(self):
        print("Woof!")
fido = Dog("Fido", "brown")
print(fido.color)
fido.bark()
```

Magic Methods/dunders

- have **double underscores** at the beginning and end of their names
- They are used to create functionality that can't be represented as a normal method.
- Common operators e.g.
    - **__str__** for changing the printing format
    - **__add__** for +
    - **__sub__** for -
    - **__mul__** for *
    - **__truediv__** for /
    - **__floordiv__** for //
    - **__mod__** for %
    - **__pow__** for **
    - **__and__** for &
    - **__xor__** for ^
    - **__or__** for |
- Comparison methods e.g.
    - **__lt__** for <
    - **__le__** for <=
    - **__eq__** for ==
    - **__ne__** for !=
    - **__gt__** for >
    - **__ge__** for >=
- Making classes act like containers
    - **__len__** for len()
    - **__getitem__** for indexing
    - **__setitem__** for assigning to indexed values
    - **__delitem__** for deleting indexed values
    - **__iter__** for iteration over objects (e.g., in for loops)
    - **__contains__** for in

Properties

- way of customizing access to instance attributes.
- created by putting the **property** decorator above a method
  - when the instance attribute with the same name as the method is accessed, the method will be called instead.
-
  - The **setter** function sets the corresponding property's value.
    - a decorator of the same name as the property, followed by a dot and the **setter** keyword.
  - The **getter** gets the value.

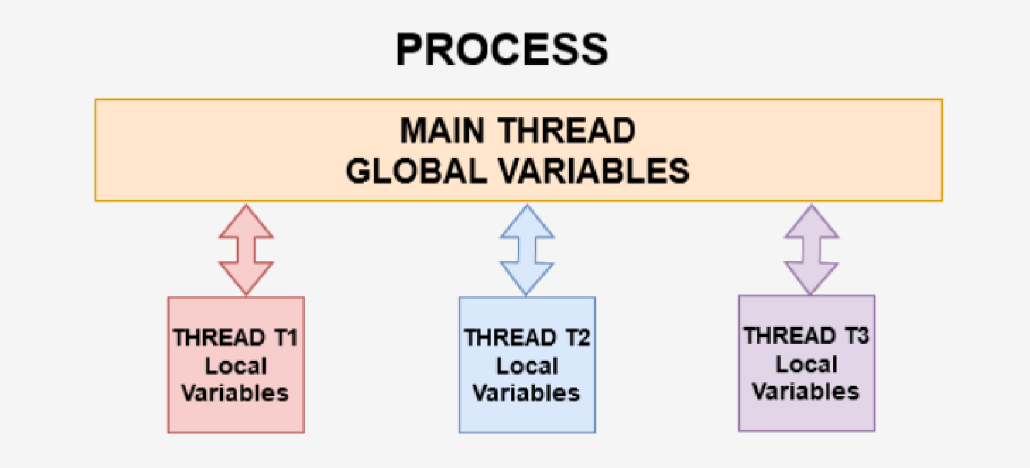
```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings
    @property
    def pineapple_allowed(self):
        return False
pizza = Pizza(["cheese", "tomato"])
print(pizza.pineapple_allowed)
pizza.pineapple_allowed = True
```

Threads

https://www.datacamp.com/tutorial/threading-in-python

- Threading allows you to have different parts of your process run concurrently
- These different parts are usually individual and have a separate unit of execution belonging to the same process
- For example, A web-browser could be a process, an application running multiple cameras simultaneously could be a process
- A thread is capable of
  - Holding data,

  - Stored in data structures like dictionaries, lists, sets, etc.
  - Can be passed as a parameter to a function.



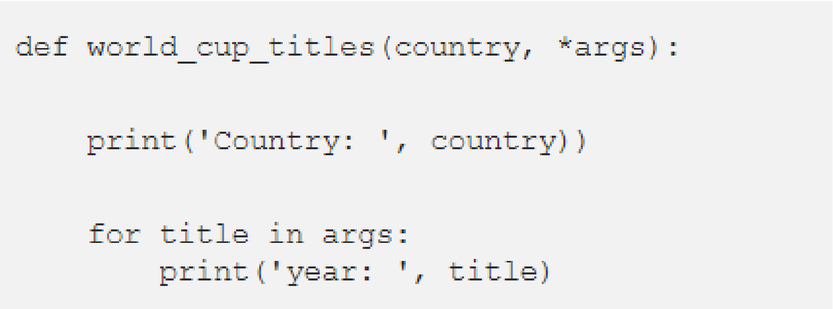
- There are different types of threads:
  - Kernel thread
  - User thread
  - Combination of kernel and user thread

```
import _thread #thread module imported
import time #time module
```

Args, Kwargs

Args

- Iteration over a list of parameters

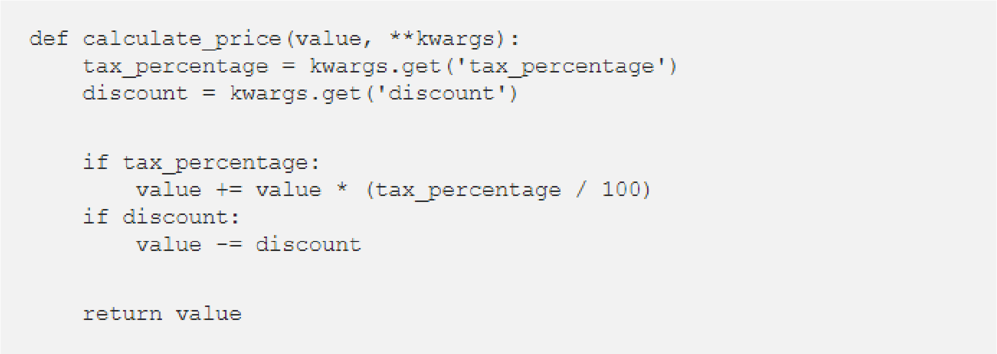


```
def world_cup_titles(country, *args):

    print('Country: ', country))

    for title in args:
        print('year: ', title)
```

Kwargs

- Different arguments for different purposes

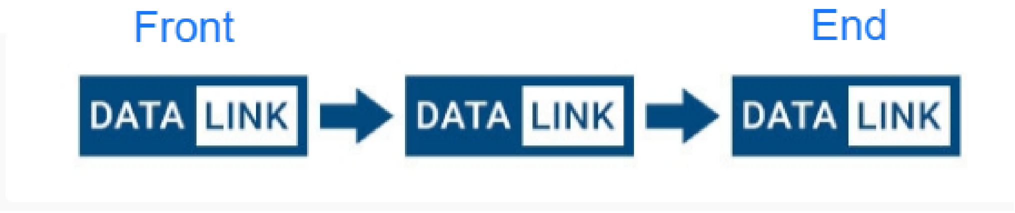
Usaremos uma função de cálculo de preço como exemplo. Nela, teremos dois argumentos opcionais, ***discount*** e ***tax_percentage:***



```
def calculate_price(value, **kwargs):
    tax_percentage = kwargs.get('tax_percentage')
    discount = kwargs.get('discount')

    if tax_percentage:
        value += value * (tax_percentage / 100)
    if discount:
        value -= discount

    return value
```

Linked Lists

- sequence of nodes where each node stores its own data and a link to the next node
- first node is called the **head**
- last node must have its link pointing to **None**

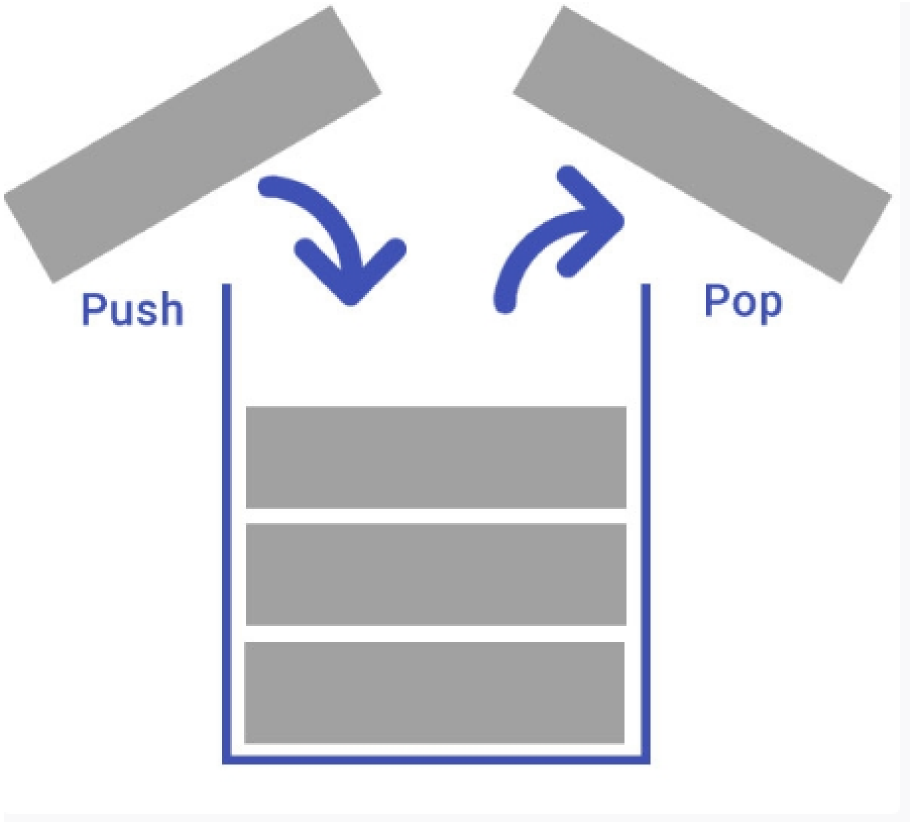


```
class Node:
    def __init__(self, data, next):
        self.data = data
        self.next = next

class LinkedList:
    def __init__(self):
        self.head = None

    def add_at_front(self, data):
        self.head = Node(data, self.head)

    def add_at_end(self, data):
        if not self.head:
            self.head = Node(data, None)
            return
        curr = self.head
        while curr.next:
            curr = curr.next
        curr.next = Node(data, None)

    def get_last_node(self):
        n = self.head
        while(n.next != None):
            n = n.next
        return n.data

    def is_empty(self):
        return self.head == None

    def print_list(self):
        n = self.head
        while n != None:
            print(n.data, end = " => ")
            n = n.next
        print()
```

Stacks and Queues

Stacks

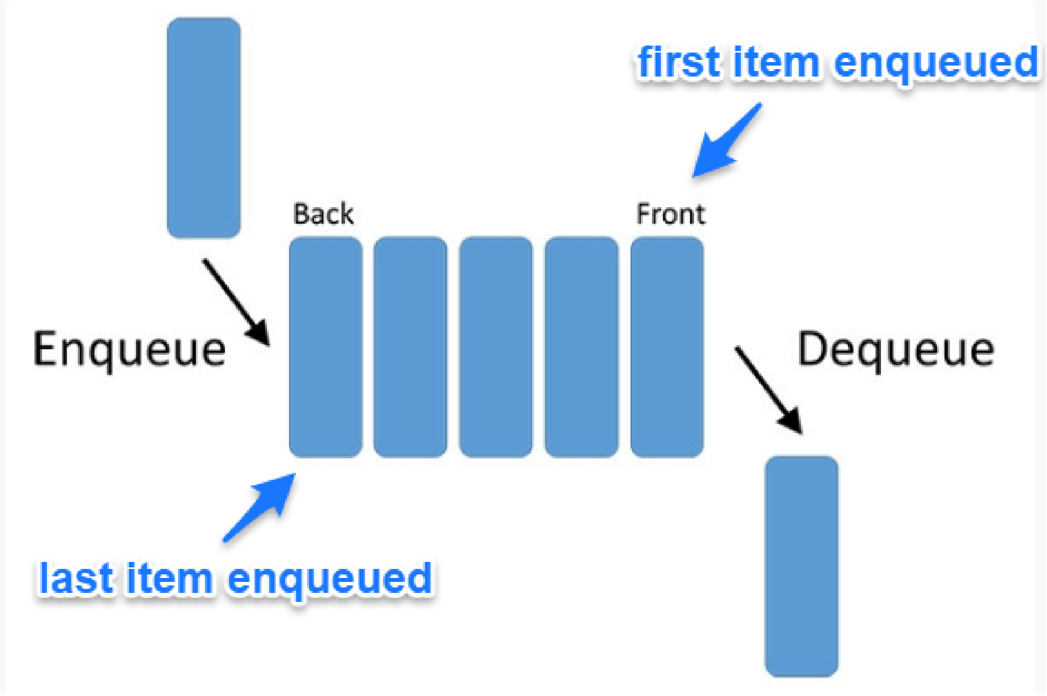
- LIFO - Last In First Out



```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.insert(0, item)

    def pop(self):
        return self.items.pop(0)

    def print_stack(self):
        print(self.items)
```

Queue

- FIFO - First In First Out



```
class Queue:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def enqueue(self, item):
        self.items.insert(0, item)
    def dequeue(self):
        return self.items.pop()
    def print_queue(self):
        print(self.items)
```

Hash Tables

https://www.youtube.com/watch?v=shs0K...

Hashtable -> key/value lookup

get a key and associate a value to look easily the data associated

image.png 442 kB

how to jump from string to index - hash function

1. takes a string
2. convert string into int
3. converts int to an index in the array

Binary Trees:

https://www.youtube.com/watch?v=Sbcii...

https://www.youtube.com/watch?v=6oL-0...

- trees are a way of path

- nodes are the points and the roots are the origin
- 3 types of traversals:
    - pre-order: Root>Left Tree>Right Tree
    - in-order:Left Tree>Root>Righ Tree
    - post-order:Left Tree>Right Tree>Root

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BinaryTree(Node):
    def __init__(self, root):
        self.root = Node(root)

    def print_tree(self, traversal_type):
        if traversal_type == 'preorder':
            return self.preorder_print(tree.root, '')

        elif traversal_type == 'inorder':
            return self.inorder_print(tree.root, '')

        elif traversal_type == 'postorder':
           return self.postorder_print(tree.root, '')

        else:
            print("traversal type " + str(traversal_type) + "not supported")
            return False


    def preorder_print(self, start, traversal):
        # Root>Left Tree>Right Tree
        if start:
            traversal += (str(start.value) + '-')
            traversal = self.preorder_print(start.left, traversal)
            traversal = self.preorder_print(start.right, traversal)

        return traversal

    def inorder_print(self, start, traversal):
        # Left Tree>Root>Right Tree
        if start:
            traversal = self.inorder_print(start.left, traversal)
            traversal += (str(start.value) + '-')
            traversal = self.inorder_print(start.right, traversal)

        return traversal

    def postorder_print(self, start, traversal):
        # Left Tree>Right Tree>Root
        if start:

            traversal = self.postorder_print(start.left, traversal)
            traversal = self.postorder_print(start.right, traversal)
            traversal += (str(start.value) + '-')

        return traversal

""""
Tree Scheme
                1
            /       \
        2              3
    /       \       /     \
  4          5    6         7
                              \
                                8
"""

tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)

tree.root.left.left = Node(4)
tree.root.left.right = Node(5)
tree.root.right.left = Node(6)
tree.root.right.right = Node(7)

tree.root.right.right.right = Node(8)

print(tree.print_tree('preorder'))
print(tree.print_tree('inorder'))
print(tree.print_tree('postorder'))
```

Depth First Search

dsdas

dsa

das

Breadth FIrst Search

dss