

Python Official Documentation: <https://docs.python.org/3/>

To visualize python code execution: <https://pythontutor.com/visualize.html#mode=edit>

Important-differences-between-python-2-x-and-python-3-x: <https://www.geeksforgeeks.org/important-differences-between-python-2-x-and-python-3-x-with-examples/> https://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html

Basics • Operators Precedence • Augmented Assignment Operator Strings • Multi line strings • Escape sequences usage • Formatted Strings List Slicing: • Most Efficient way of String Formatting • String Indexing • String Slicing: • Negative Indexing and String Reversal Built-in Functions + Methods Booleans: Type Conversion Task: Password Checker Data Structures: • Lists: o Copying Vs Modifying the List o Shallow Copy o Deep Copy o Matrix o List Methods o List Unpacking • None • Dictionary o Dictionary Keys o Dictionary Methods o Other ways of defining dictionary • Tuples • Sets: Conditional Logic • Ternary Operators(Conditional Expressions) • Logical Operators • FOR Loop • Range • Enumerate() • While Loops • break, continue, pass GUI Python end parameter in print() Functions • Parameters and Arguments(Call and Invoke) • Keyword Arguments • Default Parameters Type Hinting • Return • Docstrings • *args and *kwargs • Packing and Unpacking Arguments in Python • Scope • global keyword • nonlocal keyword Walrus Operator Advanced Python: Object Oriented Programming OOP • Attributes and Methods • Instance attributes and Class attributes • __init__ method/constructor method Duck Typing • @classmethod • @staticmethod • Instance Method • Instance vs. Static vs. Class Methods in Python 4 pillars of OOP • Encapsulation • Abstraction o Public Vs Private Variables • Inheritance o Create a Parent Class o Create a Child Class o Add the init() function o Use the super() Function o Add Propertie o Add Methods: o isinstance() • Polymorphism Super() Object Introspection Dunder Method Multiple Inheritance o Method Resolution Order(MRO)

What is the difference between run time error and compile time error in python? Explain with examples

Compile-time errors and runtime errors are two types of errors that can occur in programming, including Python.

1. Compile-Time Error:

Compile-time errors, also known as syntax errors or parsing errors, occur when the code is being compiled or translated from human-readable code to machine-executable code. These errors are detected by the compiler before the program is run. They are usually due to issues like incorrect syntax, missing or misplaced symbols, or incorrect data types.

Example:

```
print("Hello, World!"
```

In this example, the missing closing parenthesis will result in a syntax error because the code is not properly formatted.

2. Runtime Error:

Runtime errors, also known as exceptions or run-time errors, occur when the program is being executed and something unexpected happens that prevents the program from running as intended. These errors often occur due to incorrect logic, invalid inputs, or issues with the environment where the program is being run.

Example:

```
numerator = 10
denominator = 0
result = numerator / denominator
```

In this example, a ZeroDivisionError will occur at runtime because dividing by zero is not possible.

To summarize, compile-time errors are detected by the compiler during the compilation process and prevent the program from being executed at all, while runtime errors occur during the execution of the program and can cause the program to terminate or behave unexpectedly.

Python is an interpreted language, which means it doesn't have a separate compilation step like compiled languages. However, the concepts of compile-time errors (syntax errors) and runtime errors (exceptions) still apply.

Operators Precedence

<https://www.geeksforgeeks.org/python-operators/>

```
In [1]: print((5 + 4) * 10 / 2)
print(((5 + 4) * 10) / 2)
print((5 + 4) * (10 / 2))
print(5 + (4 * 10) / 2)
print(5 + 4 * 10 // 2)
```

```
45.0
45.0
45.0
25.0
25
```

Augmented Assignment Operator

```
In [2]: counter = 0
counter += 1
counter += 1
counter += 1
```

```
counter += 1
counter -= 1
counter *=2

print(counter)
```

6

Multi line strings

```
In [3]: print("hello, world i am krishna")
print('''programming :
is easy to learn if we practice regularly''')
```

```
hello, world i am krishna
programming
is easy to learn if we practice regularly
```

Escape sequences usage:

```
In [4]: print('it's too hot today')
```

```
File "<ipython-input-4-9f5a5ee85e37>", line 1
  print('it's too hot today')
  ^
SyntaxError: invalid syntax
```

```
In [17]: print("it's too hot today")
```

```
it's too hot today
```

```
In [18]: print("it's too hot"kind of" sunny")
```

```
File "<ipython-input-18-a8073202acef>", line 1
  print("it's too hot"kind of" sunny")
  ^
SyntaxError: invalid syntax
```

```
In [19]: #Usage of Escape sequence
print("it\'s too hot \"kind of\" sunny")
```

```
it's too hot "kind of" sunny
```

Formatted Strings

```
In [20]: name = 'xyz'
age =19

print('hi ' + name + ', you are ' +age+ ' old.')
```

```
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-20-580371aaca6f> in <module>
      2 age =19
      3
----> 4 print('hi ' + name + ', you are ' +age+ ' old.')

TypeError: can only concatenate str (not "int") to str
```

```
In [21]: name = 'xyz'
age =19

print('hi ' + name + ', you are ' +str(age)+ ' years old.')
```

```
hi xyz, you are 19 years old.
```

```
In [1]: name = 'xyz'
age =19
print('hi ', name, ', you are ',name, ' old.')
```

```
hi xyz , you are xyz old.
```

```
In [22]: #Formatted Strings
name = 'xyz'
age =19

print('hi {} , you are {} years old.'.format('name','age'))
```

```
hi {} , you are {} years old.
```

```
-----
AttributeError                               Traceback (most recent call last)
<ipython-input-22-e446450f6acf> in <module>
      3 age =19
      4
----> 5 print('hi {} , you are {} years old.'.format('name','age'))

AttributeError: 'NoneType' object has no attribute 'format'
```

```
In [23]: #Formatted Strings
name = 'xyz'
age =19
```

```
print('hi {} , you are {} years old.'.format(name,age))
```

hi xyz , you are 19 years old.

```
In [24]: name = 'xyz'  
age =19
```

```
print('hi {1} , you are {0} years old.'.format(name,age))
```

hi 19 , you are xyz years old.

```
In [5]: name = 'xyz'  
age =19
```

```
print('hi {new_name},you're {new_age} yrs old.'.format(new_name='abc',new_age='22'))
```

hi abc,you are 22 years old.

```
In [3]: name = 'xyz'  
age =19
```

```
print('hi {new_name},you're {age} yrs old.'.format(name,age))
```

KeyError

Traceback (most recent call last)

```
Cell In[3], line 4  
      1 name = 'xyz'  
      2 age =19  
----> 4 print('hi {new_name},you're {age} yrs old.')
```

KeyError: 'new_name'

```
In [1]: name = 'xyz'  
age =19
```

```
print('hi {new_name},you're {age} yrs old.'.format(new_name='abc',new_age='22'))
```

KeyError

Traceback (most recent call last)

```
Cell In[1], line 4  
      1 name = 'xyz'  
      2 age =19  
----> 4 print('hi {new_name},you're {age} yrs old.')
```

KeyError: 'age'

```
In [2]: name = 'xyz'  
age =19
```

```
print('hi {new_name},you're {age} yrs old.'.format(new_name='abc',age='22'))
```

hi abc,you're 22 yrs old.

f-string:

```
In [1]: #Formatted Strings  
name = 'xyz'  
age =19
```

```
print(f'hi {name} , you are {age} years old.')
```

hi xyz , you are 19 years old.

```
In [27]: print("Hello {}, your balance is {}.".format("Cindy", 50))
```

```
print("Hello {0}, your balance is {1}.".format("Cindy", 50))
```

```
print("Hello {name}, your balance is {amount}.".format(name="Cindy", amount=50))
```

```
print("Hello {0}, your balance is {amount}.".format("Cindy", amount=50))
```

Hello Cindy, your balance is 50.

Most Efficient way of String Formatting

```
In [28]: name = 'Cindy'  
amount = 50  
print(f"Hello {name}, your balance is {amount}.")
```

Hello Cindy, your balance is 50.

Format: `print("Hello {0}, your balance is {amount}.".format("Cindy", amount=50))`

f string: `print(f"Hello {name}, your balance is {amount}.")`

String Indexing:

```
In [4]: myself = 'i am learner'
        # 0123456789
x=myslef[3]
y=myslef[9]
print(x)
print(y)

m
n
```

String Slicing:

```
In [30]: myself = 'iamlearner'
        # 0123456789
#syntax=[start:stop]; we get stop-1 in output
x=myslef[3:8]
print(x)
```

learn

```
In [6]: myself = 'iamlearner'
        # 0123456789
#syntax=[start:stop:stepover]; we get stop-1 in output ---- default stepover is 1
x=myslef[3:8]
y=myslef[3:8:1]
z=myslef[3:8:2]
print(x)
print(y)
print(z)
```

learn
learn
lan

```
In [32]: myself = 'iamlearner'
        # 0123456789
#syntax=[start:stop:stepover]; we get stop-1 in output
x=myslef[3:]
y=myslef[:8]
z=myslef[::-1]
z1=myslef[::-2]
print(x)
print(y)
print(z)
print(z1)
```

learner
iamlearn
iamlearner
imere**Negative Indexing and String Reversal**

```
In [33]: myself = 'iamlearner'
        # 0123456789
#-ve index indicates to start from the end of string
x=myslef[-1]
y=myslef[::-1] # string reversal
z=myslef[::-2]
# z1=myslef[::-2]
print(x)
print(y)
print(z)
```

r
renraelmai
rnala**IMP:**

- || start index should be < stop index when stepover is positive value

- || when stepover is positive value: o/p: start to stop-1

- || start index should be > stop index when stepover is negative value

(-ve stepover means we are trying to get the values in reverse hence start index should be > stop index)

- || when stepover is negative value: o/p: start to stop+1

```
In [7]: python = 'I am PYTHON coder'
        #012345678910
print(python[3:9:1])
print(python[9:6:-1]) # 9,8,7
print(python[-1:-5:-1]) # -1,-2,-3,-4
```

m PYTH
OHT
redo

```
In [2]: str="I am Python developer" #syntax=[start:stop:stepover]; we get stop-1 in output
In [3]: str[1] # element present in the index:1
Out[3]:
'
In [4]: str[2]
Out[4]: 'a'

In [5]: str[:] # ALL the elements present in the string
Out[5]: 'I am Python developer'

In [6]: str[-1] #last element of the string
Out[6]: 'r'
```

IMP:

```
In [5]: print(str[:-2]) # starting of string to prefinal element
print(str[-2:]) # from the prefinal element
I am Pyton develop
er
In [12]: str[4:6:-1]
Out[12]:
'
In [13]: str[9:6:-1]
not
In [15]: str[-1:-6:1]
Out[15]:
'
In [18]: str[-1:-6:-1]
Out[18]: 'repol'

In [4]: python = 'I am PYTHON'
x=python+' CODER'
print(x)
I am PYTHON CODER
```

Built-in Functions + Methods

<https://docs.python.org/3/library/functions.html>

https://www.w3schools.com/python/python_ref_functions.asp

https://www.w3schools.com/python/python_ref_keywords.asp

- https://www.w3schools.com/python/ref_func_frozenset.asp
- https://www.w3schools.com/python/ref_func_reversed.asp
- https://www.w3schools.com/python/ref_func_sorted.asp

```
In [36]: python = 'I am PYTHON'
print(len(python))
11
In [37]: python = 'I am PYTHON'
print(python[0:len(python)])
print(python[0:len(python)-1])
I am PYTHON
I am PYTHO
```

Built-In Methods:

```
In [13]: python = 'I am PYTHON coder'
print(python.lower())
python = 'i will be python coder'
print(python.capitalize())
print(python.upper())
print(python.title())
print(python.find('python'))# gives the index from where the word 'python' starts
x=print(python.replace('will be','am'))
print(python) #Strings are immutable
```

```
i am python coder
I will be python coder
I WILL BE PYTHON CODER
I Will Be Python Coder
10
i am python coder
i will be python coder
```

In [39]:

```
python = 'I will be Python coder'
print(python.replace('will be','am'))
print(python) #Strings are immutable
```

```
I am Python coder
I will be Python coder
```

In [40]:

```
python = 'I will be Python coder'
print(python)
python=python.replace('will be','am')
print(python) #Strings are immutable
```

```
I will be Python coder
I am Python coder
```

all()

The all() function returns True if all items in an iterable are true, otherwise it returns False.

- If the iterable object is empty, the all() function also returns True.

In [9]:

```
mylist = [True, True, True]
x = all(mylist)
x
```

```
True
```

In [10]:

```
mylist = [0, 1, 1]
x = all(mylist)
x
```

```
False
```

In [1]:

```
myls = []
all(myls)
```

```
True
```

any()

The any() function returns True if any item in an iterable are true, otherwise it returns False.

- If the iterable object is empty, the any() function will return False.

In [11]:

```
mylist = [False, True, False]
x = any(mylist)
x
```

```
True
```

In [1]:

```
mylist=[]
x=any(mylist)
x
```

```
False
```

Booleans:

In [41]:

```
print(bool(1))
print(bool(0))
print(bool('True'))
print(bool('False'))
print(bool(()))
```

```
True
False
True
True
False
```

Type Conversion-Type Casting:

Whatever you enter as input, the input() function converts it into a string.

If you enter an integer value, still it will convert it into a string.

- If you want to number input from a user, you need to perform type conversion on the input value.

```
In [15]: birth_year=input('Enter your birth_year: ')
print(type(birth_year))
print('Your birth_year is: '+ birth_year)
age = 2020-int(birth_year)
print('your age is '+ str(age))
print(f'your age is {age}')
```

```
<class 'str'>
Your birth_year is: 2000
your age is 20
your age is 20
```

Task: Password Checker

```
In [44]: User_Name=input('Enter the User_Name: ')
Password=input('Enter the Password: ')
print(type(Password))
Hidden_Password='*' *len(Password)
print(f'Hey{User_Name}, your password {Hidden_Password} is {len(Password)} digits')
```

```
Enter the User_Name:coder
Enter the Password:coder
<class 'str'>
Hey there coder, your password ***** is 5 digits
```

```
In [45]: User_Name=input('Enter the User_Name: ')
Password=input('Enter the Password: ')
#print(type(Password))
print(f'{User_Name}, your password {'*'*len(Password)} is {len(Password)} digits')
```

```
Enter the User_Name:coder
Enter the Password:coder
coder, your password ***** is 5 digits
```

```
In [46]: print(type('a'))
```

```
<class 'str'>
```

Data Structures:

Lists:

```
In [2]: amazon_cart=['books','pens']
amazon_cart[0]
```

```
Out[2]: 'books'
```

My Finding:

```
In [3]: 4*amazon_cart
```

```
Out[3]: ['books', 'pens', 'books', 'pens', 'books', 'pens', 'books', 'pens']
```

```
In [4]: amazon_cart+amazon_cart
```

```
Out[4]: ['books', 'pens', 'books', 'pens']
```

```
In [5]: amazon_cart+(2*amazon_cart)
```

```
Out[5]: ['books', 'pens', 'books', 'pens', 'books', 'pens']
```

```
In [6]: amazon_cart-amazon_cart
```

```
-----
TypeError                                 Traceback (most recent call last)
Cell In[6], line 1
----> 1 amazon_cart-amazon_cart

TypeError: unsupported operand type(s) for -: 'list' and 'list'
```

```
In [29]: list=[1,2,3]
2*list
```

```
Out[29]: [1, 2, 3, 1, 2, 3]
```

```
In [48]: amazon_cart
```

```
Out[48]: ['books', 'pens']
```

- Strings are immutable. i.e., we can't change the values
- Lists are mutable. i.e., we can change the values

List Slicing:

```
In [49]: string='abcdefg'
string[0]
```

```
Out[49]: 'a'
```

```
In [50]: string[0]='x' #Strings are immutable. i.e., we can't change the values
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-50-033609a963b2> in <module>
      1 string[0]='x' #Strings are immutable. i.e., we can't change the values
----> 2     TypeError: 'str' object does not support item assignment
```

```
In [8]: amazon_cart=['books','pens','toys','food']
amazon_cart[0:4:1]
```

```
Out[8]: ['books', 'pens', 'toys', 'food']
```

```
In [9]: amazon_cart[0:3:2]
```

```
Out[9]: ['books', 'toys']
```

```
In [10]: amazon_cart[0]='papers' #Lists are mutable. i.e., we can change the values
```

```
In [11]: amazon_cart
```

```
Out[11]: ['papers', 'pens', 'toys', 'food']
```

```
In [12]: new_cart=amazon_cart[0:4]
```

```
new_cart[1]='tins'
print(new_cart)
print(amazon_cart)
```

```
['papers', 'tins', 'toys', 'food']
['papers', 'pens', 'toys', 'food']
```

Copying Vs Modifying the List:

<https://www.youtube.com/watch?v=SgUwPDT9tEs>

```
In [16]: amazon_cart=['books','pens','toys','food']
amazon_cart[0]='papers'
new_cart=amazon_cart
new_cart[1]='tins'
print(new_cart)
print(amazon_cart)
```

```
#Here amazon_cart got modified, it contains 'tins' which we never assigned....
#Something is fishy right?
```

```
['papers', 'tins', 'toys', 'food']
['papers', 'tins', 'toys', 'food']
```

always use list slicing to copy the list

```
In [57]: amazon_cart=[
    'books',
    'pens',
    'toys',
    'food'
]
amazon_cart[0]='papers'
new_cart=amazon_cart[:] #always use list slicing to copy the list
new_cart[1]='tins'
print(new_cart)
print(amazon_cart)
```

```
['papers', 'tins', 'toys', 'food']
['papers', 'pens', 'toys', 'food']
```

new_cart=amazon_cart[:] --- always use list slicing to copy the list

- amazon_cart[:] --- will create a new copy of the list and that will be assigned to new_Cart

```
In [17]: lst1=[1,2,3,4]
lst2=lst1
print(lst2)
print(lst1)
```

```
print(id(lst2)) # id() --- gives the address
print(id(lst1))
```

```
# both list1, list2 are pointing to the same memory address, hence any change in list2 will also reflect in list1
```

```
[1, 2, 3, 4]
[1, 2, 3, 4]
2077876821696
2077876821696
```

In [18]:

```
lst3=[1,2,3,4]
lst4=lst3[:]
print(lst3)
print(lst4)

print(id(lst3))
print(id(lst4))

# List3, List4 are pointing to different memory address, hence any change in List4 will not reflect in List3
```

```
[1, 2, 3, 4]
[1, 2, 3, 4]
2077878474752
2077878478592
```

`lst2=lst1`

- both list1, list2 are pointing to the same memory address, hence any change in list2 will also reflect in list1

`lst4=lst3[:]`

- list3, list4 are pointing to different memory address, hence any change in list4 will not reflect in list3

'=' operation --- Assignment operator

In [10]:

```
lst1=[1,2,3,4]
lst2=lst1
print(lst2[2])
lst2[2]=10
print(lst2)
print(lst1)
```

```
3
[1, 2, 10, 4]
[1, 2, 10, 4]
```

In [9]:

```
lst1=[1,2,3,4]
print(f'address of lst1 is: {id(lst1)}')
```

```
address of lst1 is: 2382961210816
```

`id()` gives the unique ID or address where the variable or object is stored in the memory.

In [3]:

```
lst1=[1,2,3,4]
print(f'addr of lst1 is: {id(lst1)}')
lst2=lst1
print(f'addr of lst2 is: {id(lst2)}')
print(lst2[2])
lst2[2]=10
print(lst2)
print(lst1)
```

```
addr of lst1 is: 2246368252352
addr of lst2 is: 2246368252352
3
[1, 2, 10, 4]
[1, 2, 10, 4]
```

`When assignment operator is used, lst2 will point to the same address of lst1. Hence whatever changes we make in lst2 will be reflected in lst1 also.`

Shallow Copy

In [11]:

```
# Shallow copy
lst1=[1,2,3,4]
lst2=lst1.copy()
print(lst2[2])
lst2[2]=10
print(lst2)
print(lst1) #lst1 is not modified
```

```
3
[1, 2, 10, 4]
[1, 2, 3, 4]
```

In [4]:

```
# Shallow copy
lst1=[1,2,3,4]
print(f'addr of lst1 is: {id(lst1)}')
lst2=lst1.copy()
print(f'addr of lst2 is: {id(lst2)}')
print(lst2[2])
```

```
lst2[2]=10
print(lst2)
print(lst1) #lst1 is not modified

addr of lst1 is: 2246368254784
addr of lst2 is: 2246368255296
3
[1, 2, 10, 4]
[1, 2, 3, 4]
```

`lst1.copy()` will create a new copy of the `lst1` and that will be assigned to `lst2`

In [7]: `# Another approach of Shallow copy`

```
import copy
lst1=[1,2,3,4]
lst2=copy.copy(lst1)
print(lst2[2])
lst2[2]=10
print(lst2)
print(lst1) #lst1 is not modified

3
[1, 2, 10, 4]
[1, 2, 3, 4]
```

In [11]: `# Shallow copy`

```
lst1=[[1,2,3,4],[5,6,7]] #nested List
lst2=lst1.copy()
print(lst2)
print(lst2[0][0])
lst2[1][0]=190
print(lst2)
print(lst1) #lst1 is modified --hence to avoid this in nested list deep copy is used

[[1, 2, 3, 4], [5, 6, 7]]
1
[[1, 2, 3, 4], [190, 6, 7]]
[[1, 2, 3, 4], [190, 6, 7]]
```

In the above example, `lst1` is modified --- shallow copy is not applicable for nested lists.

- In nested list `deep copy` is used

Deep Copy:

In [13]: `# Deep copy`

```
import copy
lst1=[10,20,30,40]
lst2=copy.deepcopy(lst1)
print(lst2[2])
lst2[2]=100
print(lst2)
print(lst1) #lst1 is not modified

30
[10, 20, 100, 40]
[10, 20, 30, 40]
```

- for non-nested list, shallow copy works same as deep copy

In [15]: `# Deep copy`

```
lst1=[[11,21,31,41],[51,61,71]]
lst2=copy.deepcopy(lst1)
print(lst2)
print(lst2[1][0])
lst2[1][0]=191
print(lst2)
print(lst1) #lst1 is not modified

[[11, 21, 31, 41], [51, 61, 71]]
51
[[11, 21, 31, 41], [191, 61, 71]]
[[11, 21, 31, 41], [51, 61, 71]]
```

`Shallow copy` works for non-nested lists

`Deep copy` works for non-nested lists and nested lists

Exercise on List:

In [58]:

```
new_list = ['a', 'b', 'c']
print(new_list[1])
print(new_list[-2])
```

```
print(new_list[1:3])
new_list[0] = 'z'
print(new_list)
```

```
my_list = [1,2,3]
bonus = my_list + [5]
my_list[0] = 'z'
print(my_list)
print(bonus)
```

```
b
b
['b', 'c']
['z', 'b', 'c']
['z', 2, 3]
[1, 2, 3, 5]
```

In [59]:

```
new_list = ['a', 'b', 'c']
print(new_list)
new_list[1]='d'
print(new_list)
```

```
['a', 'b', 'c']
['a', 'd', 'c']
```

Matrix:

In [60]:

```
matrix=[  
    [1,0,1],  
    [0,1,0],  
    [1,0,1]  
]
```

In [61]:

```
matrix
```

Out[61]:

```
[[1, 0, 1], [0, 1, 0], [1, 0, 1]]
```

In [62]:

```
matrix[0][1]=5
print(matrix)
```

```
[[1, 5, 1], [0, 1, 0], [1, 0, 1]]
```

In [63]:

```
matrix1=[  
    [[1],0,1],  
    [0,1,0],  
    [1,0,1]  
]
```

Multidimensional matrix:

In [64]:

```
matrix1
```

Out[64]:

```
[[[1], 0, 1], [0, 1, 0], [1, 0, 1]]
```

In [65]:

```
print(matrix1[0][0])
print(matrix1[0][0][0])
```

```
[1]
```

```
1
```

List & Matrix:

My Findings:

In [66]:

```
print(new_list)
new_list[1]='d'
print(new_list)

print(matrix)
matrix[0][1]=4
print(matrix)
```

```
['a', 'd', 'c']
['a', 'd', 'c']
[[1, 5, 1], [0, 1, 0], [1, 0, 1]]
[[1, 4, 1], [0, 1, 0], [1, 0, 1]]
```

In [67]:

```
new_list = ['a', 'b', 'c']
print(new_list)
new_list[1]='d'
print(new_list)

matrix=[[1,0,1],[0,1,0],[1,0,1]]
print(matrix)
matrix[0][1]=5
print(matrix)
```

```
['a', 'b', 'c']
['a', 'd', 'c']
[[1, 0, 1], [0, 1, 0], [1, 0, 1]]
[[1, 5, 1], [0, 1, 0], [1, 0, 1]]
```

Exercise:

```
In [68]: # using this list:
basket = [
    "Banana",
    ["Apples", ["Oranges"], "Blueberries"]
];
# access "Oranges" and print it:
print(basket[1][1])
print(basket[1][1][0])

print(basket[0])
```

```
['Oranges']
Oranges
Banana
```

```
In [69]: matrix1=[[[1],0,1],[0,1,0],[1,0,1]]
print(matrix1[0][0])
print(matrix1[0][0][0])

# using this list:
basket = [
    ["Banana", "grapes"],
    ["Apples", ["Oranges"], "Blueberries"]
]
# access "Oranges" and print it:
print(basket[1][1])
print(basket[1][1][0])
```

```
[1]
1
['Oranges']
Oranges
```

Semicolons in Python:

Semicolons have very little use in Python, and the use of them is not considered "Pythonic." Nevertheless, they typically do not do anything, as one who has moved from a language such as C or Java may observe. ... The semicolon does have use in Python, however. It is used to separate commands on a single line.

Python does let you use a semi-colon to denote the end of a statement if you are including more than one statement in a line. Semicolons can be used to one line two or more commands. They don't have to be used, but they aren't restricted.

Use a semi-colon to denote the end of a statement if you are including more than one statement in a line.

```
In [1]: basket = [
    ["Banana", "grapes"],
    ["Apples", "Oranges", "Blueberries"]
] print(basket[1][2])
```

```
Cell In[1], line 4
 ] print(basket[1][2])
 ^
SyntaxError: invalid syntax
```

```
In [20]: basket = [
    ["Banana", "grapes"],
    ["Apples", "Oranges", "Blueberries"]
]; print(basket[1][2])
```

```
Blueberries
```

```
In [19]: basket = [
    ["Banana", "grapes"],
    ["Apples", "Oranges", "Blueberries"]
]
print(basket[1][2])
```

```
Blueberries
```

List Methods:

https://www.w3schools.com/python/python_ref_list.asp

```
In [71]: basket=[1,2,3,4,5]
print(len(basket))
```

5

- `append` modifies the list inplace.

```
In [72]: basket=[1,2,3,4,5]
new_list=basket.append(100)
print(basket)
print(new_list) #append modifies the list inplace.
#i.e., basket.append(100) only appends 100 to basket
```

[1, 2, 3, 4, 5, 100]
None

```
In [3]: basket=[1,2,3,4,5]
print(basket.append(100))
```

None

```
In [4]: basket=[1,2,3,4,5]
basket.append(100)
print(basket)
```

[1, 2, 3, 4, 5, 100]

```
In [73]: basket=[1,2,3,4,5]
basket.append(100)
new_list=basket
print(new_list)
```

[1, 2, 3, 4, 5, 100]

```
In [74]: basket=[1,2,3,4,5]
basket.insert(0,10) #basket.insert(index,value)
new_list=basket
print(new_list)
```

[10, 1, 2, 3, 4, 5]

```
In [75]: basket=[1,2,3,4,5]
basket.extend([100,101])
new_list=basket
print(new_list)
```

[1, 2, 3, 4, 5, 100, 101]

- `pop(index)`: removes the element in the mentioned index
- `pop()`: removes the last element when index is not defined

```
In [76]: basket=[1,2,3,4,5]
basket.extend([100,101])
basket.pop() # removes the last element when index is not defined
print(basket)
```

[1, 2, 3, 4, 5, 100]

```
In [77]: basket=[1,2,3,4,5]
basket.extend([100,101])
print(basket)
basket.pop()# removes the last element
print(basket)
basket.pop()# removes the last element
print(basket)
```

[1, 2, 3, 4, 5, 100, 101]
[1, 2, 3, 4, 5, 100]
[1, 2, 3, 4, 5]

```
In [78]: basket=[1,2,3,4,5]
basket.pop(0)# removes the element in the mentioned index
print(basket)
```

[2, 3, 4, 5]

```
In [79]: basket=[1,2,3,4,5]
basket.remove(4)# removes the value mentioned
print(basket)
```

[1, 2, 3, 5]

Difference between `pop()` & `remove()`:

- `pop(index)` --- removes the element in the mentioned index
- `remove(element)` --- removes the element

```
In [3]: basket=[1,2,3,4,5]
basket.pop(2) #pop(index)
print(basket)
basket=[1,2,3,4,5]
```

```
basket.remove(4) #remove(element)
print(basket)
```

```
[1, 2, 4, 5]
[1, 2, 3, 5]
```

```
In [5]: basket=[1,2,3,4,5]
basket.clear()
print(basket)

[]
```

`basket.append()`, `basket.insert()`, `basket.extend([])`, `basket.remove()`, `basket.clear()` : These methods doesn't return anything --- These methods does inplace modifications/changes

`basket.pop()` : This method returns the value that it removed from the list

```
In [22]: basket=[1,2,3,4,5]
new_list=basket.append(100)
print(new_list)
print(basket) # Inplace modification

basket=[1,2,3,4,5]
new_list=basket.insert(0,10)
print(new_list)
print(basket) # Inplace modification

basket=[1,2,3,4,5]
new_list=basket.extend([100,101])
print(new_list)
print(basket) # Inplace modification

basket=[1,2,3,4,5]
new_list=basket.pop(0)
print(new_list)
print(basket) # Inplace modification

basket=[1,2,3,4,5]
new_list=basket.remove(4)
print(new_list)
print(basket) # Inplace modification

basket=[1,2,3,4,5]
new_list=basket.clear()
print(new_list)
print(basket) # Inplace modification
```

```
None
[1, 2, 3, 4, 5, 100]
None
[10, 1, 2, 3, 4, 5]
None
[1, 2, 3, 4, 5, 100, 101]
1
[2, 3, 4, 5]
None
[1, 2, 3, 5]
None
[]
```

```
In [7]: basket=[1,2,3,4,5]
new_list=basket.extend(100,101) # extend usually takes one iterable
print(new_list)
```

```
-----  
TypeError  
Cell In[7], line 2
    1 basket=[1,2,3,4,5]
----> 2 new_list=basket.extend(100,101)
    3 print(new_list)
```

Traceback (most recent call last)

`TypeError: list.extend() takes exactly one argument (2 given)`

```
In [1]: basket=[1,2,3,4,5]
basket.append(100)
print(basket)

basket=[1,2,3,4,5]
basket.insert(0,10)
print(basket)

basket=[1,2,3,4,5]
basket.extend([100,101])
print(basket)

basket=[1,2,3,4,5]
basket.append('hello')
print(basket)

basket=[1,2,3,4,5]
basket.insert(0,'hello')
```

```
print(basket)

basket2=[1,2,3,4,5]
basket2.extend('hello')
print(basket2)

[1, 2, 3, 4, 5, 100]
[10, 1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 100, 101]
[1, 2, 3, 4, 5, 'hello']
['hello', 1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 'h', 'e', 'l', 'l', 'o']
```

Difference between extend() and append():

```
In [16]: basket=[1,2,3,4,5]
basket.extend([100,101])
print(basket)

basket2=[1,2,3,4,5]
basket2.extend('hello')
print(basket2)

basket2=[1,2,3,4,5]
basket2.extend(13)
print(basket2)

[1, 2, 3, 4, 5, 100, 101]
[1, 2, 3, 4, 5, 'h', 'e', 'l', 'l', 'o']
```

TypeError Traceback (most recent call last)
Cell In[16], line 10
7 print(basket2)
9 basket2=[1,2,3,4,5]
---> 10 basket2.extend(13)
11 print(basket2)

TypeError: 'int' object is not iterable

```
In [15]: basket=[1,2,3,4,5]
basket.append('hello')
print(basket)

basket=[1,2,3,4,5]
basket.append(100)
print(basket)

basket=[1,2,3,4,5]
basket.append(['hello',12])
print(basket)

basket=[1,2,3,4,5]
basket.append(100,1)
print(basket)

[1, 2, 3, 4, 5, 'hello']
[1, 2, 3, 4, 5, 100]
[1, 2, 3, 4, 5, ['hello', 12]]
```

TypeError Traceback (most recent call last)
Cell In[15], line 15
11 print(basket)
14 basket=[1,2,3,4,5]
---> 15 basket.append(100,1)
16 print(basket)

TypeError: list.append() takes exactly one argument (2 given)

index(value,start,stop)

- Here start and stop values are where to start and stop looking in the list.
- starts looking at start index value and stops at stop-1

```
In [4]: basket=['a','b','c','d']
print(basket.index('c'))
```

2

```
In [4]: basket=['a','b','c','d']
print(basket.index('c',0,1))

# Here the interpreter start looking at index 0 and stops looking at index=0 but c is present at index2
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[4], line 2
  1 basket=['a','b','c','d']
----> 2 print(basket.index('c',0,1))

ValueError: 'c' is not in list
```

```
In [5]: basket=['a','b','c','d']
print(basket.index('c',0,2))
#starts looking at start index value and stops at stop-1
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[5], line 2
  1 basket=['a','b','c','d']
----> 2 print(basket.index('c',0,2))

ValueError: 'c' is not in list
```

```
In [15]: basket=['a','b','c','d']
print(basket.index('c', 0, 4))
```

2

```
In [16]: basket=['a','b','c','d']
print('d' in basket)
```

True

```
In [18]: basket=['a','b','c','d']
print ('x' in basket)
```

False

```
In [19]: print('i' in 'this story is purely based on imagination')
```

True

```
In [20]: basket=['a','b','c','d']
print (basket.count('d')) #count gives the number of times a value occurs
```

1

```
In [21]: basket=['a','b','c','d','d']
print (basket.count('d'))
```

2

Exercise:

```
In [35]: # using this list,
basket = ["Banana", "Apples", "Oranges", "Blueberries"];

# 1. Remove the Banana from the list

# 2. Remove "Blueberries" from the list.

# 3. Put "Kiwi" at the end of the list.

# 4. Add "Apples" at the beginning of the list

# 5. Count how many apples in the basket

# 6. empty the basket

basket = ["Banana", "Apples", "Oranges", "Blueberries"];
basket.remove('Banana')
print(basket)
basket.pop()
print(basket)
basket.append("Kiwi")
print(basket)
basket.insert(0,"Apples")
print(basket)
print(basket.count("Apples"))
basket.clear()
```

```
['Apples', 'Oranges', 'Blueberries']
['Apples', 'Oranges']
['Apples', 'Oranges', 'Kiwi']
['Apples', 'Apples', 'Oranges', 'Kiwi']
2
```

```
In [33]: print(basket)
```

[]

Difference between sort() and sorted():

https://www.w3schools.com/python/ref_list_sort.asp

https://www.w3schools.com/python/ref_func_sorted.asp

- `sort()` --- sorts the list Inplace.
- `sorted()` --- this creates a new copy of list and sorts. `sorted(list)` --- gives a new sorted list.
 - `ls.sort(reverse=True)` --- descending order
 - `ls.sort(reverse=False)` --- by default `reverse=False` --- ascending order

```
In [17]: basket=[8,5,7,12,3,9,7,1,3,5,4]
basket.sort()
print(basket.sort()) #sorts the list - Inplace modification
print(basket)
```

None
[1, 3, 3, 4, 5, 5, 7, 7, 8, 9, 12]

```
In [37]: basket=[8,5,7,12,3,9,7,1,3,5,4]

print(sorted(basket)) # this creates a new copy of basket and sorts

print(basket)

[1, 3, 3, 4, 5, 5, 7, 7, 8, 9, 12]
[8, 5, 7, 12, 3, 9, 7, 1, 3, 5, 4]
```

```
In [39]: # sorted builtin function equals to:
basket=[8,5,7,12,3,9,7,1,3,5,4]

new_basket=basket[:]
new_basket.sort()
print(new_basket)

print(basket)

[1, 3, 3, 4, 5, 5, 7, 7, 8, 9, 12]
[8, 5, 7, 12, 3, 9, 7, 1, 3, 5, 4]
```

```
In [40]: # sorted builtin function equals to:
basket=[8,5,7,12,3,9,7,1,3,5,4]

new_basket=basket.copy() #copies the list and returns the new list
new_basket.sort()
print(new_basket)

print(basket)

[1, 3, 3, 4, 5, 5, 7, 7, 8, 9, 12]
[8, 5, 7, 12, 3, 9, 7, 1, 3, 5, 4]
```

```
In [5]: ls = [8,5,7,12,3,9,7,1,3,5,4]
ls.sort()
print(ls)
```

[1, 3, 3, 4, 5, 5, 7, 7, 8, 9, 12]

```
In [4]: ls = [8,5,7,12,3,9,7,1,3,5,4]
ls.sort(reverse=True) # descending order
print(ls)
```

[12, 9, 8, 7, 7, 5, 5, 4, 3, 3, 1]

```
In [11]: a = ("h", "b", "a", "c", "f", "d", "e", "g")
x = sorted(a) #by default reverse=False # ascending order
print(x)
```

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

```
In [9]: a = ("h", "b", "a", "c", "f", "d", "e", "g")
x = sorted(a, reverse=True)
print(x)
```

['h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']

```
In [12]: a = ("h", "b", "a", "c", "f", "d", "e", "g")
x = sorted(a, reverse=False)
print(x)
```

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

```
In [41]: basket=[8,5,7,12,3,9,7,1,3,5,4]
basket.reverse() #reverses the list
print(basket)
```

[4, 5, 3, 1, 7, 9, 3, 12, 7, 5, 8]

```
In [43]: basket=[8,5,7,12,3,9,7,1,3,5,4]
basket.sort()
print(basket)
basket.reverse() #reverses the list - Inplace
print(basket)
```

```
[1, 3, 3, 4, 5, 5, 7, 8, 9, 12]
[12, 9, 8, 7, 5, 4, 3, 3, 1]
```

```
In [44]: basket=[8,5,7,12,3,9,7,1,3,5,4]
print(len(basket))
```

```
11
```

```
In [50]: basket=[8,5,7,12,3,9,7,1,3,5,4]
print(basket[::-1]) #reverse and creates a new list
print(basket)
```

```
[4, 5, 3, 1, 7, 9, 3, 12, 7, 5, 8]
[8, 5, 7, 12, 3, 9, 7, 1, 3, 5, 4]
```

Difference between reverse() and reversed():

- `list.reverse()` --- reverses the list inplace
- `reversed(list)` --- gives an reverse iterator object

```
In [1]: basket=[8,5,7,12,3,9,7,1,3,5,4]
print(basket.reverse()) #reverses the list inplace
print(basket)
```

```
None
```

```
[4, 5, 3, 1, 7, 9, 3, 12, 7, 5, 8]
```

https://www.w3schools.com/python/ref_func_reversed.asp

```
In [9]: basket=[8,5,7,12,3,9,7,1,3,5,4]
print(reversed(basket)) # gives an reverse iterator object
lst=[]
for i in reversed(basket):
    lst.append(i)
print(lst)
```

```
<list_reverseiterator object at 0x000001DDC2CE1DE0>
[4, 5, 3, 1, 7, 9, 3, 12, 7, 5, 8]
```

Exercise:

```
In [23]: #fix this code so that it prints a sorted list of all of our friends (alphabetical).
friends = ['Simon', 'Patty', 'Joy', 'Carrie', 'Amira', 'Chu']

new_friend = ['Stanley']

print(friends.sort() + new_friend)
```

```
-----
TypeError                                     Traceback (most recent call last)
Cell In[23], line 6
      2 friends = ['Simon', 'Patty', 'Joy', 'Carrie', 'Amira', 'Chu']
      3 new_friend = ['Stanley']
----> 6 print(friends.sort() + new_friend)
```

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'list'
```

```
print(friends.sort() + new_friend)
```

gives error because `friends.sort()` returns None as `.sort()` does inplace modification

```
In [58]: friends = ['Simon', 'Patty', 'Joy', 'Carrie', 'Amira', 'Chu']
new_friend = ['Stanley']
friends.extend(new_friend)
print(friends)
friends.sort()
print(friends)
```

```
['Simon', 'Patty', 'Joy', 'Carrie', 'Amira', 'Chu', 'Stanley']
['Amira', 'Carrie', 'Chu', 'Joy', 'Patty', 'Simon', 'Stanley']
```

Range():

```
In [17]: print(range(1,100)) #range(start value, stop value, step)
print(list(range(1,100))) #prints a list with start, stop-1 values
print(list(range(100)))
```

```
range(1, 100)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66,
67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 3
4, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 6
6, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 9
8, 99]
```

join():

```
In [25]: #join()
symbol='!'
new_sentence=symbol.join('hi','this','is','pratice')
print(new_sentence)
```

```
-----
TypeError                                     Traceback (most recent call last)
Cell In[25], line 3
  1 #join()
  2 symbol='!'
----> 3 new_sentence=symbol.join('hi','this','is','pratice')
  4 print(new_sentence)

TypeError: str.join() takes exactly one argument (4 given)
```

```
In [26]: #join()
symbol='!'
sent = symbol.join(['hi','this','is','practice'])
print(sent)
```

```
hi!this!is!practice
```

```
In [24]: symbol='!'
new_sentence=symbol.join(['hi','this','is','pratice'])
#joins the entire list with the given symbol
print(new_sentence)
```

```
symbol=''
new_sentence=symbol.join(['hi','this','is','pratice'])
#joins the entire list with the given symbol
print(new_sentence)
```

```
symbol=' '
new_sentence=symbol.join(['hi','this','is','pratice'])
#joins the entire list with the given symbol
print(new_sentence)
```

```
sentence=' '.join(['hi','this','is','pratice'])
print(sentence)
```

```
sentence=' '.join(['hi','this','is','pratice'])
print(sentence)
```

```
hi!this!is!pratice
hithisispratice
hi this is pratice
hithisispratice
hi this is pratice
```

```
In [7]: x='#'
print(x.join(['hey','there','hello']))
print('OS'.join(['hey','there','hello']))
print(' '.join(['hey','there','hello']))
```

```
hey#there#hello
heyOSthereOShello
hey there hello
```

List Unpacking:

```
In [6]: list=[1,2,3]
print(list)
print(list[0])
```

```
[1, 2, 3]
1
```

```
In [4]: a,b,c=[1,2,3] #assigning each value in a list to each variable
```

```
print(a,b,c)
```

```
print(a)
```

```
print(b)
```

```
print(c)
```

```
1 2 3
1
2
3
```

```
In [5]: #list unpacking
a,b,c, *others=[1,2,3,4,5,6,7,8,9] #assigning each value in a list to each variable
```

```
print(a)
```

```
print(b)
```

```
print(c)
```

```
print(others)
```

```

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

In [6]: a,b,c,*others,d=[1,2,3,4,5,6,7,8,9] #assigning each value in a list to each variable
print(a)
print(b)
print(c)
print(others)
print(d)

```

```

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

```

```

In [27]: *hello=[1,2,3,4,5,6,7,8,9]
print(hello)

File "<ipython-input-27-75bf5b27c5e0>", line 4
SyntaxError: starred assignment target must be in a list or tuple

```

```

In [62]: hello=[1,2,3,4,5,6,7,8,9]
print(*hello)

```

```
1 2 3 4 5 6 7 8 9
```

```

In [63]: hello=[1,2,3,4,5,6,7,8,9]
print(*hello[2])

```

```

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-63-3a16418c240a> in <module>
      1 hello=[1,2,3,4,5,6,7,8,9]
----> 2 print(*hello[2])

TypeError: print() argument after * must be an iterable, not int

```

```

In [8]: hello=[1,2,3,4,5,6,7,8,9]
print([*hello][2]) #accesing an elemnt after unpacking the list completely.

```

```
3
```

Important Finding:

```

In [16]: *hello=[1,2,3,4,5,6,7,8,9]
print(hello)

Cell In[16], line 1
  *hello=[1,2,3,4,5,6,7,8,9]
  ^
SyntaxError: starred assignment target must be in a list or tuple

```

```

In [13]: hello=[1,2,3,4,5,6,7,8,9]
print(*hello)

```

```
1 2 3 4 5 6 7 8 9
```

```

In [14]: hello=[1,2,3,4,5,6,7,8,9]
print(*hello[2])

-----
TypeError                                 Traceback (most recent call last)
Cell In[14], line 2
      1 hello=[1,2,3,4,5,6,7,8,9]
----> 2 print(*hello[2])

TypeError: print() argument after * must be an iterable, not int

```

```

In [15]: hello=[1,2,3,4,5,6,7,8,9]
print([*hello][2])

```

```
3
```

```

In [9]: [*others] = [1,2,3,4]
print(others)

*others1, = [1,2,3,4]
print(others1)

others2= [1,2,3,4]
print(*others2)

```

```
[1, 2, 3, 4]
[1, 2, 3, 4]
1 2 3 4
```

```

In [67]: hello=[1,2,3,4,5,6,7,8,9]
print(*hello)#unpacking the list completely.
print([*hello][2]) #accesing an elemnt after unpacking the list completely.
print([*hello][8])

```

```
1 2 3 4 5 6 7 8 9
3
9
```

```
In [41]: lst=['one', 'one', 'two', 'three', 'one', 'two']
{x:lst.count(x) for x in lst}
```

```
Out[41]: {'one': 3, 'two': 2, 'three': 1}
```

None:

```
In [5]: weapons= None
print(weapons)
```

```
None
```

Dictionary:

```
In [18]: dict= {
    'a':1,
    'b':2
}
# Dictionary is unordered key:value pair...
#unordered means that they are not right next to each other in memory
```

```
In [19]: print(dict['b'])
#A key is a string for us to grab the value what we need.
```

```
2
```

```
In [11]: print(dict['c'])
```

```
-----
KeyError                                                 Traceback (most recent call last)
<ipython-input-11-c69c8779aef8> in <module>
----> 1 print(dict['c'])

KeyError: 'c'
```

```
In [12]: print(dict)
```

```
{'a': 1, 'b': 2}
```

I actually return the entire dictionary maybe I don't have this in order these are all just scattered all across our memory as you see here when I receive the dictionary I get these things in order but that's only because it's small if I had a really really large dictionary I might not have them in order that I've inserted so a dictionary is an unordered key value pair and as long as we know the key that is whatever the key that we're looking for then we just give that and our computer is going to know hey where in memory to look to grab the values.

dictionary is an unordered data structure but list is ordered, we can access the values in the list by their index as they are stored in the memory in an order.

```
In [20]: dict= {
    'a':[1,2,3],
    'b':"Hello",
    'c':True
}           # dictionary having list, string and boolean values
```

```
In [15]: print(dict)
{'a': [1, 2, 3], 'b': 'Hello', 'c': True}
```

```
In [16]: print(dict['a'])
[1, 2, 3]
```

```
In [17]: print(dict['a'][0])
1
```

```
In [23]: my_list= [
    {
        'a':[1,2,3],
        'b':"Hello",
        'c':True
    },
    {
        'a':[4,5,6],
        'b':"Hi",
        'c':False
    }
]      # list containing dictionary
```

```
In [24]: print(my_list[0])
{'a': [1, 2, 3], 'b': 'Hello', 'c': True}
```

IMP:

when list contains dict, we needs to access the elements using the key

```
In [25]: print(my_list[0]['a'])
```

[1, 2, 3]

```
In [26]: print(my_list[0]['a'][2])
```

3

Dictionary Keys

```
In [19]: dictionary = {
    123:[1,2,3],           #dictionary can have strings and int as keys
    'b':"Hello",
    'c':True
}
```

```
In [20]: print(dictionary[123])
```

[1, 2, 3]

```
In [21]: dictionary = {
    123:[1,2,3],
    True:"Hello",          #dictionary can have boolean as keys
    'c':True
}
```

```
In [22]: print(dictionary[True])
```

Hello

```
In [23]: dictionary = {
    123:[1,2,3],
    True:"Hello",          # dictionary can't have List as keys
    [100]:True
}
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-23-31aa0e1a4f33> in <module>
      2     123:[1,2,3],
      3     True:"Hello",          #dictionary can't have list as keys
----> 4     [100]:True
      5

TypeError: unhashable type: 'list'
```

Note:

dictionary can't have list as keys

Dictionary key needs to be immutable (not changable)

Dictionary can have strings, numbers, booleans as keys but dictionary can't have list as keys. Mostly dictionary will have strings as keys

key in dictionary need to be unique....if we use the same key then the value will be overwritten.

```
In [1]: dictionary = {
    123:[1,2,3],
    123:"Hello",
    #key in dictionary need to be unique....
    #if we use the same key then the value will be overwritten.
    'c':True
}
```

```
In [2]: print(dictionary[123])
```

Hello

Other ways of defining dictionary:

```
In [1]: user= dict(name='developer', age=25)
#dict() is a built-in function which will create an empty dictionary
print(user)

{'name': 'developer', 'age': 25}
```

```
In [2]: user=dict()
user['greet']='hello'; user['name']='developer'
print(user)

{'greet': 'hello', 'name': 'developer'}
```

Dictionary Methods:

https://www.w3schools.com/python/python_ref_dictionary.asp

```
In [3]: dictionary= {
    'basket':[1,2,3],
    'greet':"Hello",
    'c':True
}

In [4]: print(dictionary['basket'])
[1, 2, 3]

In [5]: print(dictionary.copy())
{'basket': [1, 2, 3], 'greet': 'Hello', 'c': True}

In [6]: print(dict['age'])

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-6-df37446b5b13> in <module>
      1 print(dict['age'])
----> 1 print(dict['age'])

TypeError: 'type' object is not subscriptable

In [30]: print(dictionary.get('age'))
# get is a method on the object or the dictionary in Python.

None
```

If let's say there's no age (key) in the user but you want to have a default value.

you add a comma and then here you say whatever you want as the default value

```
In [7]: print(dictionary.get('age', 55))
55

In [8]: print(dictionary)
{'basket': [1, 2, 3], 'greet': 'Hello', 'c': True}
```

```
In [9]: dictionary= {
    'basket':[1,2,3],
    'greet':"Hello",
    'c':True
}

In [10]: print(dictionary.get('age'))
None

In [13]: print(dictionary.get('age', 55))
55

In [12]: print(dictionary.get('age'))
None

In [14]: dictionary
Out[14]: {'basket': [1, 2, 3], 'greet': 'Hello', 'c': True}
```

```
In [9]: dictionary= {
    'basket':[1,2,3],
    'greet':"Hello",
    'age':20
}

In [10]: print(dictionary.get('age', 55))
# here age is mentioned in the dict it doesn't take the default value..

20

In [3]: dictionary= {
    'basket':[1,2,3],
    'greet':"Hello",
    'age':20
}
```

```
In [6]: print('basket' in dictionary)
print('size' in dictionary)

True
False

In [7]: print('basket' in dictionary.keys())

True

In [8]: print('Hello' in dictionary.values())

True

In [10]: print(dictionary.items()) #Returns a list containing a tuple for each key value pair
dict_items([('basket', [1, 2, 3]), ('greet', 'Hello'), ('age', 20)])

In [11]: dictionary= {
    'basket':[1,2,3],
    'greet':"Hello",
    'age':20
}

In [12]: dictionary.clear()

In [13]: print(dictionary)

{}

In [17]: dictionary= {
    'basket':[1,2,3],
    'greet':"Hello",
    'age':20
}
dictionary2=dictionary.copy()
print(dictionary)
print(dictionary2)

{'basket': [1, 2, 3], 'greet': 'Hello', 'age': 20}
{'basket': [1, 2, 3], 'greet': 'Hello', 'age': 20}
```

IMP:

```
In [19]: dictionary= {
    'basket':[1,2,3],
    'greet':"Hello",
    'age':20
}
dictionary2=dictionary.copy()
dictionary.clear()
print(dictionary)
print(dictionary2)

{}
{'basket': [1, 2, 3], 'greet': 'Hello', 'age': 20}
```

In [20]:

```
dd= {
    'basket':[1,2,3],
    'greet':"Hello",
    'age':20
}

dd['age']
```

Out[20]:

20

In [21]:

```
dd= {
    'basket':[1,2,3],
    'greet':"Hello",
    'age':20
}

dd.get('age')
```

Out[21]:

20

In [22]:

```
x = ('key1', 'key2', 'key3')
y = 0

thisdict = dict.fromkeys(x, y)

print(thisdict)

{'key1': 0, 'key2': 0, 'key3': 0}
```

In [23]:

```
x = ('key1', 'key2', 'key3')
y = 0,2,3
```

```
thisdict = dict.fromkeys(x, y)
print(thisdict)

{'key1': (0, 2, 3), 'key2': (0, 2, 3), 'key3': (0, 2, 3)}
```

- `pop(key)`: removes the value present in the key and returns that value

```
In [4]: user= {
    'basket':[1,2,3],
    'greet':"Hello",
    'age':20
}

print(user.pop('age'))
```

20

```
In [5]: print(user)

{'basket': [1, 2, 3], 'greet': 'Hello'}
```

- `popitem()`: removes the last key:value pair and returns that pair

```
In [6]: print(user.popitem())
print(user)

('greet', 'Hello')
{'basket': [1, 2, 3]}
```

`update():`

- value gets updated if key is present in the dictionary
- value gets added if key is not present in the dictionary

```
In [7]: user= {
    'basket':[1,2,3],
    'greet':"Hello",
    'age':20
}
print(user.update({'age':55}))
```

*#check the syntax properly
#age values gets updated if age is present in the dictionary*

`print(user)`

None

{'basket': [1, 2, 3], 'greet': 'Hello', 'age': 55}

```
In [31]: dictionary= {
    'basket':[1,2,3],
    'greet':"Hello",
    'age':20
}
print(dictionary.update({'ages':55}))
#ages values gets added as it is not present in the dictionary
print(dictionary)
```

None

{'basket': [1, 2, 3], 'greet': 'Hello', 'age': 20, 'ages': 55}

```
In [34]: dictionary= {
    'basket':[1,2,3],
    'greet':"Hello",
    'c':True
}
```

```
In [38]: print(dictionary['age'])
```

```
-----
KeyError                                                 Traceback (most recent call last)
<ipython-input-38-16413cf84b7a> in <module>
      1 print(dictionary['age'])
----> 1 print(dictionary['age'])

KeyError: 'age'
```

`setdefault():`

- value gets added if the key is not present in the dictionary
- nothing happens, if the key is already present in the dictionary

```
In [15]: d= {
    'basket':[1,2,3],
    'greet':"Hello",
    'c':True
```

```

        }
d
{'basket': [1, 2, 3], 'greet': 'Hello', 'c': True}

```

In [10]: `d['age']`

```

-----
KeyError                                     Traceback (most recent call last)
Cell In[10], line 1
----> 1 d['age']

KeyError: 'age'

```

In [11]: `d.setdefault('age',65) #age value gets added as it is not present in the dictionary`
`print(d['age'])`
`print(d)`

```

65
{'basket': [1, 2, 3], 'greet': 'Hello', 'c': True, 'age': 65}

```

In [40]: `dictionary.setdefault('age',65)`
`print(dictionary)`

```

{'basket': [1, 2, 3], 'greet': 'Hello', 'c': True, 'age': 65}

```

In [15]: `d = {`
 `'basket':[1,2,3],`
 `'greet':'Hello',`
 `'age':65,`
 `'c':True`
`}`
`d`

Out[15]: `{'basket': [1, 2, 3], 'greet': 'Hello', 'age': 65, 'c': True}`

In [16]: `d.setdefault('age',75)`
`print(d)`

```

{'basket': [1, 2, 3], 'greet': 'Hello', 'age': 65, 'c': True}

```

Exercise:

In []: `#1 Create a user profile for your new game.`
`#This user profile will be stored in a dictionary`
`#with keys: 'age', 'username', 'weapons', 'is_active' and 'clan'`

`#2 iterate and print all the keys in the above user.`

`#3 Add a new weapon to your user`

`#4 Add a new key to include 'is_banned'. Set it to false`

`#5 Ban the user by setting the previous key to True`

`#6 create a new user2 by copying the previous user and`
`#update the age value and username value.`

In [6]: `user={`
 `'age':21,`
 `'username':'gamer',`
 `'weapons':'coding',`
 `'is_active':'Yes',`
 `'clan':'India'`
`}`
`print(user)`

```

{'age': 21, 'username': 'gamer', 'weapons': 'coding', 'is_active': 'Yes', 'clan': 'India'}

```

In [1]: `user={`
 `'age':21,`
 `'username':'gamer',`
 `'weapons':'coding',`
 `'is_active':'Yes',`
 `'clan':'India'`
`}`
`user['check'] = 'hey'`
`print(user)`

```

{'age': 21, 'username': 'gamer', 'weapons': 'coding', 'is_active': 'Yes', 'clan': 'India', 'check': 'hey'}

```

In [7]: `print(user.keys())`
`dict_keys(['age', 'username', 'weapons', 'is_active', 'clan'])`

In [10]: `user.update({'weapons':'skills'})`
`print(user)`

```

{'age': 21, 'username': 'gamer', 'weapons': 'skills', 'is_active': 'Yes', 'clan': 'India'}

```

In [11]: `user.update({'is_banned':False})`
`print(user)`

```
{'age': 21, 'username': 'gamer', 'weapons': 'skills', 'is_active': 'Yes', 'clan': 'India', 'is_banned': False}
```

```
In [12]: user.update({'is_banned':True})
print(user)
```

```
{'age': 21, 'username': 'gamer', 'weapons': 'skills', 'is_active': 'Yes', 'clan': 'India', 'is_banned': True}
```

```
In [9]: user2=user.copy()
print(user2)
```

```
{'age': 28, 'username': 'coder', 'weapons': 'coding', 'is_active': 'Yes', 'clan': 'India', 'check': 'hey'}
```

```
In [10]: user2.update({'age':28,'username':'coder'})
print(user2)
```

```
{'age': 28, 'username': 'coder', 'weapons': 'coding', 'is_active': 'Yes', 'clan': 'India', 'check': 'hey'}
```

```
In [19]: # Solutions:
```

```
# 1 Create a user profile for your new game.
```

```
#This user profile will be stored in a dictionary with keys:
```

```
#'age', 'username', 'weapons', 'is_active' and 'clan'
```

```
user = {
    'age': 22,
    'username': 'Shogun',
    'weapons': ['katana', 'shuriken'],
    'is_active': True,
    'clan': 'Japan'
}
```

```
}
```

```
print(user)
```

```
# 2 iterate and print all the keys in the above user.
```

```
print(user.keys())
```

```
# 3 Add a new weapon to your user
```

```
user['weapons'].append('shield')
```

```
print(user)
```

```
# 4 Add a new key to include 'is_banned'. Set it to false
```

```
user.update({'is_banned': False})
```

```
print(user)
```

```
# 5 Ban the user by setting the previous key to True
```

```
# user['is_banned'] = True
```

```
print(user)
```

```
# 6 create a new user2 my copying the previous user
```

```
#and update the age value and username value.
```

```
user2 = user.copy()
```

```
user2.update({'age': 100, 'username': 'Timbo'})
```

```
print(user2)
```

```
{'age': 22, 'username': 'Shogun', 'weapons': ['katana', 'shuriken'], 'is_active': True, 'clan': 'Japan'}
```

```
dict_keys(['age', 'username', 'weapons', 'is_active', 'clan'])
```

```
{'age': 22, 'username': 'Shogun', 'weapons': ['katana', 'shuriken', 'shield'], 'is_active': True, 'clan': 'Japan'}
```

```
{'age': 22, 'username': 'Shogun', 'weapons': ['katana', 'shuriken', 'shield'], 'is_active': True, 'clan': 'Japan', 'is_banned': False}
```

```
False
```

```
{'age': 22, 'username': 'Shogun', 'weapons': ['katana', 'shuriken', 'shield'], 'is_active': True, 'clan': 'Japan', 'is_banned': False}
```

```
False
```

```
{'age': 100, 'username': 'Timbo', 'weapons': ['katana', 'shuriken', 'shield'], 'is_active': True, 'clan': 'Japan', 'is_banned': False}
```

```
False
```

```
{'age': 100, 'username': 'Timbo', 'weapons': ['katana', 'shuriken', 'shield'], 'is_active': True, 'clan': 'Japan', 'is_banned': False}
```

```
False
```

```
In [17]: user={
```

```
    'name':"coder",
    "age":22
}
```

```
}
```

```
for x in user:
```

```
    print(x) #prints keys
```

```
for x in user.items():
```

```
    print(x)
```

```
for x in user.values():
```

```
    print(x)
```

```
for x in user.keys():
```

```
    print(x)
```

```
name
```

```
age
```

```
('name', 'coder')
```

```
('age', 22)
```

```
coder
```

```
22
```

```
name
```

```
age
```

```
In [18]: user={
```

```
    'name':"coder",
    "age":22
}
```

```
}
```

```
print(user.items())
```

```
for x in user.items():
```

```
    print(x)
```

```
dict_items([('name', 'coder'), ('age', 22)])
```

```
('name', 'coder')
```

```
('age', 22)
```

```
In [19]: user={
    'name':"coder",
    "age":22
}
for i in user.items():
    key,value=i;
    print(key,value)
```

name coder
age 22

```
In [1]: user={
    'name':"coder",
    "age":22
}
for i in user.items():
    key,value=i;
    print(key,value)
```

name coder
age 22

```
In [3]: user={
    'name':"coder",
    "age":22
}
for key,value in user.items():
    print(key,value)
```

name coder
age 22

```
In [19]: user={
    'name':"coder",
    "age":22
}
for key,value in user.items():
    print(key)
```

name
age

```
In [18]: user={
    'name':"coder",
    "age":22
}
for key,value in user.items():
    print(value)
```

coder
22

Tuples:

Tuples are immutable lists

```
In [1]: my_tuple=(1,2,3,4,5)
print(my_tuple)
print(my_tuple[0])
#we can access the values in the tuples using their index same like the lists.
print(5 in my_tuple)
my_tuple[1]='x'
print(my_tuple)
```

(1, 2, 3, 4, 5)
1
True

```
-----  
TypeError                                     Traceback (most recent call last)  
Cell In[1], line 6  
      4 #we can access the values in the tuples using their index same like the lists.  
      5 print(5 in my_tuple)  
----> 6 my_tuple[1]='x'  
      7 print(my_tuple)
```

TypeError: 'tuple' object does not support item assignment

So a good use of a tuple is for example if you work at Uber is a geographic location and coordinates right.

You can have latitude and longitude here and because let's say this won't change or maybe a user's location or pickup point doesn't often change we can just use a tuple.

However maybe when you're in a car as a driver and you're Uber driver well you probably need your coordinates your latitude longitude and a list because your car is moving that's constantly changing.

```
In [26]: user = {
    'age': 22,
    'username': 'coder',
    'weapons': ['katana', 'shuriken'],
```

```

    'is_active': True,
    'clan': 'India'
}
print(user.items()) #returns key:value pairs as tuples

dict_items([('age', 22), ('username', 'coder'), ('weapons', ['katana', 'shuriken']), ('is_active', True), ('clan', 'India')])

In [5]: user = {
    (1,2): [1,2,3],    #tuples can be used as keys in dictionaries
    'clan': 'India'
}
print(user[(1,2)])

```

[1, 2, 3]

```

In [32]: user = {
    (1,2): [1,2,3],
    'clan': 'India'
}
print(user[(1,2)][2])

```

3

Tuple Methods:

https://www.w3schools.com/python/python_ref_tuple.asp

```

In [38]: my_tuple=(1,2,3,4,5)
my_tup2=my_tuple[1:2]
my_tup3=my_tuple[1:3]
print(my_tup2)
print(my_tup3)

```

(2,
2, 3)

```

In [2]: my_tuple=(1,2,3,4,5)
my_tup2=my_tuple[1:7]
print(my_tup2)

mylist=[1,2,3,4,5]
mylist2=mylist[1:7]
print(mylist2)

```

(2, 3, 4, 5)
[2, 3, 4, 5]

```

In [41]: my_tuple=(1,2,3,4,5)
x=my_tuple[1]
y=my_tuple[2]
print(x,y)

```

2 3

```

In [42]: x,y,*others=(1,2,3,4,5)
print(x,y,others)

```

1 2 [3, 4, 5]

```

In [47]: my_tuple=(1,2,3,4,5,5,5)
        #0,1,2,3,4,5,6
print(my_tuple.count(5))
print(my_tuple.index(5))
#Well the index of 5 is 4 because ,
#for the first value that it finds it's going to return the index of.
print(len(my_tuple))

```

3
4
7

Sets:

Unordered collection of unique objects

https://www.w3schools.com/python/python_ref_set.asp

```

In [48]: #Unordered collection of unique objects
my_set={1,2,3,4,5}
print(my_set)

```

{1, 2, 3, 4, 5}

```

In [49]: my_set={1,2,3,4,5,5}
print(my_set)  # only returns the unique values.i.e., not duplications

```

{1, 2, 3, 4, 5}

```

In [50]: my_set={1,2,3,4,5,5}
my_set.add(100)

```

```
my_set.add(1)
print(my_set)

{1, 2, 3, 4, 5, 100}

In [9]: my_list=[1,2,3,4,5,5,5]
set(my_list)      #we can remove duplicates from the List using set()

Out[9]: {1, 2, 3, 4, 5}
```

Note:

set is not subscriptable

```
In [2]: my_set={1,2,3,4,5,5}
print(my_set[1])
#The error is indicating that the function or method is not subscriptable;
# means they are not indexable like a list or sequence.
```

```
-----  
TypeError  
Cell In[2], line 2  
    1 my_set={1,2,3,4,5,5}  
----> 2 print(my_set[1])
```

Traceback (most recent call last)

```
TypeError: 'set' object is not subscriptable
```

```
In [3]: print(1 in my_set)
print(len(my_set)) #only counts the unique things.
print(list(my_set))

True
5
[1, 2, 3, 4, 5]
```

```
In [10]: my_set={1,2,3,4,5,5}
new_set=my_set.copy()
print(new_set)

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

```
In [12]: my_set={1,2,3,4,5,5}
new_set=my_set.copy()
print(new_set)
my_set.clear()
print(my_set)

{1, 2, 3, 4, 5}
set()
```

```
In [46]: set1={1,2,3,4,5}
set2={4,5,6,7,8,9}
set1.difference(set2)
```

```
Out[46]: {1, 2, 3}
```

```
In [44]: set1={1,2,3,4,5}
set2={4,5,6,7,8,9}
print(set1.difference(set2))
print(set1)
print(set2)

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

```
In [2]: set1={1,2,3,4,5}
set2={4,5,6,7,8,9}
set1.difference_update(set2)      #removes the differences and updates set1
print(set1)
print(set2)

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

```
In [39]: set1={1,2,3,4,5}
set2={4,5,6,7,8,9}
set1.discard(5)
print(set1)

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

```
In [2]: set1={1,2,3,4,5}
set2={4,5,6,7,8,9}
set1.intersection(set2)
```

```
Out[2]: {4, 5}
```

```
In [10]: set1={1,2,3,4,5}
set2={4,5,6,7,8,9}
set1&set2 #intersection
```

Out[10]: {4, 5}

```
In [4]: set1={1,2,3,4,5}
set2={4,5,6,7,8,9}
set1.isdisjoint(set2)#is disjoint means the 2 sets doesn't have any intersection
```

Out[4]: False

```
In [5]: set1={1,2,3}
set2={4,5,6,7,8,9}
set1.isdisjoint(set2)
```

Out[5]: True

```
In [7]: set1={1,2,3,4,5}
set2={4,5,6,7,8,9}
set1.union(set2)
```

Out[7]: {1, 2, 3, 4, 5, 6, 7, 8, 9}

```
In [9]: set1={1,2,3,4,5}
set2={4,5,6,7,8,9}
set1|set2 #union
```

Out[9]: {1, 2, 3, 4, 5, 6, 7, 8, 9}

```
In [11]: set1={4,5}
set2={4,5,6,7,8,9}
set1.issubset(set2)
```

Out[11]: True

```
In [15]: set1={4,5}
set2={4,5,6,7,8,9}
set1.issuperset(set2)
```

Out[15]: False

```
In [14]: set1={4,5}
set2={4,5,6,7,8,9}
set2.issuperset(set1)
```

Out[14]: True

Discard(), Remove(), Pop():

```
In [3]: fruits = {"apple", "banana", "cherry"}
fruits.discard("banana")
print(fruits)
```

{'cherry', 'apple'}

```
In [4]: fruits = {"apple", "banana", "cherry"}
fruits.discard("mango")
print(fruits)
```

{'cherry', 'apple', 'banana'}

```
In [5]: fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")
print(fruits)
```

{'cherry', 'apple'}

```
In [6]: fruits = {"apple", "banana", "cherry"}
fruits.remove("mango")
print(fruits)
```

KeyError

Traceback (most recent call last)

```
Cell In[6], line 2
      1 fruits = {"apple", "banana", "cherry"}
----> 2 fruits.remove("mango")
      3 print(fruits)
```

KeyError: 'mango'

The remove() method will raise an error if the specified item does not exist, and the discard() method will not.

```
In [8]: fruits = {"apple", "banana", "cherry"}
fruits.pop() #removes a random item from the set.
print(fruits)
```

{'apple', 'banana'}

Exercise:

```
In [ ]: # You are working for the school Principal. We have a database of school students:
school = {'Bobby', 'Tammy', 'Jammy', 'Sally', 'Danny'}

#during class, the teachers take attendance and compile it into a list.
attendance_list = ['Jammy', 'Bobby', 'Danny', 'Sally']

'''using what you learned about sets, create a piece of code that the
school principal can use to
immediately find out who missed class so they can call the parents.
Imagine if the list had 1000s of students.
The principal can use the lists generated by the teachers + the school database
to use python & make his/her job easier'''
#Find the students that miss class!
```

```
In [32]: # Solution: Notice how we don't have to convert the attendance_list to a set
#...it does it for you.
print(school.difference(attendance_list))

{'Tammy'}
```

Conditional Logic:

```
In [10]: x= True
if x:
    print("its true")
print("Check")

its true
Check
```

```
In [11]: x= False
if x:
    print("its true")
print("Check")

Check
```

```
In [12]: x= False
if x:
    print("its true")
else:
    print("Check")

Check
```

```
In [13]: x= True
if x:
    print("its true")
else:
    print("Check")

its true
```

```
In [14]: x= False
y=True
if x:
    print("its true")
elif y:
    print("ok")
else:
    print("Check")

ok
```

```
In [15]: x= False
y=False
if x:
    print("its true")
elif y:
    print("ok")
else:
    print("Check")

Check
```

```
In [17]: x= True
y=True
if x and y:
    print("Perfect")
else:
    print("Check")
```

Perfect

In python, indentation in a loop should have 4 spaces as per standard

```
In [18]: x= 'Hello'
y=5
```

```
print(bool("Hello"))
print(bool(5))
if x and y:
    print("Perfect")
else:
    print("Check")
```

True
True
Perfect

In [20]:

```
x= ''
y=0
print(bool(""))
print(bool(0))
if x and y:
    print("Perfect")
else:
    print("Check")
```

False
False
Check

Ternary Operators(Conditional Expressions):

In [11]:

```
'''syntax:condition_if_true if condition else condition_if_else
if condition is true.....condition_if_true will execute
if condition is false.....condition_if_else will execute'''

is_frnd=True
can_msg="msg allowed" if is_frnd else "not allowed to msg"
print(can_msg)

is_frnd=False
can_msg="msg allowed" if is_frnd else "not allowed to msg"
print(can_msg)
```

msg allowed
not allowed to msg

Short circuiting: By short circuiting we mean the stoppage of execution of boolean operation if the truth value of expression has been determined already. The evaluation of expression takes place from left to right. In python, short circuiting is supported by various boolean operators and functions.

or: When the Python interpreter scans or expression, it takes first statement and checks to see if it is true. If the first statement is true, then Python returns that object's value without checking the second statement. The program does not bother with the second statement. If the first value is false, only then Python checks the second value and then result is based on second half. and: For an and expression, Python uses a short circuit technique to check if the first statement is false then the whole statement must be false, so it returns that value. Only if the first value is true, it checks the second statement and returns the value. An expression containing and or stops execution when the truth value of expression has been achieved. Evaluation takes place from left to right.

In [27]:

```
#Short circuiting
x= True
y=False
if x or y:
    print("Perfect")
else:
    print("Check")
```

Perfect
Check

Logical Operators:

and,or,<,>,==,>=,<=,!~,not

In [37]:

```
print(4>5)
print(4==5) # checking equality
print(not(4==5))
```

False
False
True

In [32]:

```
print(4=5) #this means assigment
```

```
File "<ipython-input-32-5c8ee04f8d4e>", line 1
    print(4=5)
      ^
SyntaxError: keyword can't be an expression
```

```
In [36]: print('a'>'b') #ascii value of a is < ascii value of b
print('a' > 'A') #ascii value of a is > ascii value of A

False
True
```

```
In [45]: is_magician=False
is_expert=True
if (not(is_magician) and is_expert):
    print("You are a master magician")

You are a master magician
```

```
In [8]: is_magician=False
is_expert=False
if (not(is_magician) and not(is_expert)):
    print("Atleast you are getting there")
```

Atleast you are getting there

```
In [5]: is_magician=False
is_expert=True
if not is_magician:
    print("You need Magical powers")
```

You need Magical powers

Difference between ==, is, ===:

Operators:

1. '==' checks the equality of both the values
2. 'is' actually checks if the location/memory where both values are stored is the same.
3. '===' check if the location where both values are stored and the data types of both of them is the same.

```
In [9]: print(True==1)
print(''==1)
print([]==1)
print(10==10.0)
print([]==[]) # == checks the equality of both the values
```

True
False
False
True
True

```
In [7]: print(True is 1)
print('' is 1)
print([] is 1 )
print(10 is 10.0)
print([] is [])
#is actually checks if the location and memory where both values are stored is the same.
```

```
False
False
False
False
False
<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:2: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:3: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:4: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:2: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:3: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:4: SyntaxWarning: "is" with a literal. Did you mean "=="?
<ipython-input-7-aaa5f47834f9>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
    print(True is 1)
<ipython-input-7-aaa5f47834f9>:2: SyntaxWarning: "is" with a literal. Did you mean "=="?
    print('' is 1)
<ipython-input-7-aaa5f47834f9>:3: SyntaxWarning: "is" with a literal. Did you mean "=="?
    print([] is 1 )
<ipython-input-7-aaa5f47834f9>:4: SyntaxWarning: "is" with a literal. Did you mean "=="?
    print(10 is 10.0)
```

```
In [8]: print(True is True)
print('1' is '1')
print([] is [])
```

True
True
False

```
<>:2: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:2: SyntaxWarning: "is" with a literal. Did you mean "=="?
<ipython-input-8-ff49cff56559>:2: SyntaxWarning: "is" with a literal. Did you mean "=="?
    print('1' is '1')
```

Every time I create a list it's added in memory somewhere.

So this is in a location in memory but whenever I create a new list it's created in another location. So these are two completely different lists that live in different locations in memory. so it's going to check hey is this in the same memory space same bookshelf as that one.'

```
In [12]: print([1,2,3] is [1,2,3]) #memory check
print([1,2,3] == [1,2,3]) #values check
```

False
True

Exercise: (try and except)

```
In [4]: largest = None
smallest = None
while True:
    num = input("Enter a number: ")
    if num == "done":
        break
    num = int(num)
    if largest is None or num > largest:
        largest = num
    elif smallest is None or num < smallest:
        smallest = num
print ("Maximum is", largest)
print ("Minimum is", smallest)
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[4], line 7
      5     if num == "done":
      6         break
----> 7     num = int(num)
      8     if largest is None or num > largest:
      9         largest = num

ValueError: invalid literal for int() with base 10: 'ij'
```

```
In [5]: largest = None
smallest = None
while True:
    try:
        num = input("Enter a number: ")
        if num == "done":
            break
        num = int(num)
        if largest is None or num > largest:
            largest = num
        elif smallest is None or num < smallest:
            smallest = num
    except ValueError:
        print("Invalid input")
        continue

print ("Maximum is", largest)
print ("Minimum is", smallest)
```

Invalid input
Invalid input
Maximum is 778
Minimum is 0

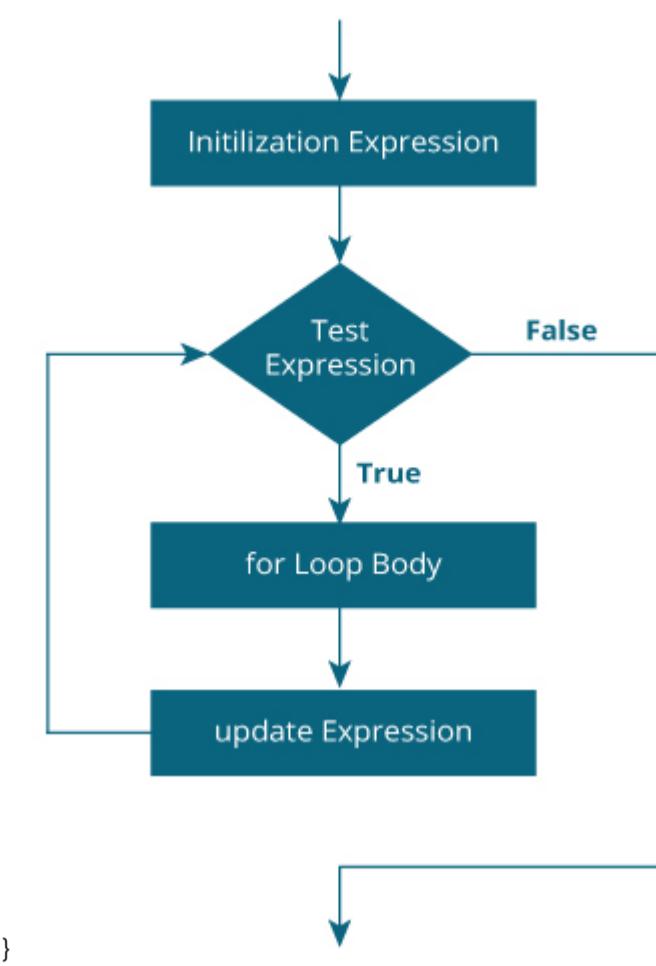
FOR Loop:

for loops allow us to iterate over anything that has a collection of items

lists,dictionary,tuple,set,strings are iterable. iterated means that we can go one by one to check each item

```
for (initializationStatement; testExpression; updateStatement) {

    // statements inside the body of loop
```



```
In [23]: for item in "Zero":
    print(item)
```

Z
e
r
o

```
In [3]: for item in 5:
    print("hi")
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-3-5fb632b798e1> in <module>
      1 for item in 5:
      2     print("hi")
-----
TypeError: 'int' object is not iterable
```

```
In [34]: for item in [1,2,3]:
    print(item)
```

1
2
3

```
In [39]: user={
    'name':'coder',
    'age':22
}
for item in user:
    print(item) #prints keys
for item in user.items():
    print(item)
for item in user.values():
    print(item)
for item in user.keys():
    print(item)
```

name
age
('name', 'coder')
('age', 22)
coder
22
name
age

```
In [45]: user={
    'name':'coder',
    'age':22
}
for item in user.item():
    key,value=item;
    print(key,value)
```

```

-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-45-b5e784a21ad5> in <module>
      3     "age":22
      4 }
----> 5 for item in user.item():
      6     key,value=item;
      7     print(key,value)

AttributeError: 'dict' object has no attribute 'item'

```

```
In [14]: user={
    'name':"coder",
    "age":22
}
for item in user.items():
    key,value=item;
    print(key,value)
```

name coder
age 22

```
In [6]: user={
    'name':"coder",
    "age":22
}
for key,value in user.items():
    print(key,value)
```

name coder
age 22

```
In [6]: user={
    'name':"coder",
    "age":22
}
for k, v in user.items():
    print(key,value)
```

```

-----
NameError                                     Traceback (most recent call last)
Cell In[6], line 6
      1 user={
      2     'name':"coder",
      3     "age":22
      4 }
      5 for k, v in user.items():
----> 6     print(key,value)

NameError: name 'key' is not defined

```

The below code block was executed after restarting the kernel... here key,value are not having any values as the kernel is restarted and if the kernel is restarted all the variables will be lost. and as key, value are not defined here it throws error.

```
In [2]: user={
    'name':"coder",
    "age":22
}
for k, v in user.items():
    print(k,v)
```

name coder
age 22

```
In [31]: for item in [1,2]:
    print(item)
    print(item)
```

1
1
2
2

```
In [33]: for item in [1,2]:
    for x in ['a','b']:
        print(item,x)
```

1 a
1 b
2 a
2 b

Exercise:

```
In [2]: my_list = [1,2,3,4,5]
counter = 0
for i in my_list:
    counter = counter + i
print(counter)
```

15

```
In [2]: my_list = [1,2,3,4,5]
counter = 0
for item in my_list:
    counter = counter + item
    print(counter)
print(counter)

1
3
6
10
15
15
```

```
In [2]: my_list = [1,2,3,4,5]
counter = 0
i=5
for item in my_list:
    counter = counter + i
    print(counter)

5
10
15
20
25
```

Range:

```
In [7]: print(range(10))
print(range(0, 10))

range(0, 10)
range(0, 10)
```

```
In [6]: for i in range(0, 5):
    print (i)

0
1
2
3
4
```

```
In [8]: for _ in range(0, 3):
    #Range(start,stop,step)., start value included and stop value excluded
    print (_)

0
1
2
```

In python programs, If a programmer doesn't need variable you just do an underscore and this is just a variable. I mean you could use it like this and you'll still get these numbers but it just indicates to other people that hey I don't really care what this variable is.

```
In [9]: for _ in range(0, 10,2):
    print(_)

0
2
4
6
8
```

```
In [10]: for i in range(0, 5,-1):
    print (i) # this wont work and we wont get any output
```

```
In [11]: for i in range(5,0):
    print (i) # this wont work and we wont get any output
```

```
In [13]: for i in range(3,0,-1):
    # if we want to print in reverse then we should use -1 as step.
    print (i)

3
2
1
```

```
In [14]: for i in range(2):
    print(list(range(0,3)))

[0, 1, 2]
[0, 1, 2]
```

Enumerate():

A lot of times when dealing with iterators, we also get a need to keep a count of iterations. Python eases the programmers' task by providing a built-in function `enumerate()` for this task. `Enumerate()` method adds a counter to an iterable and returns it in a form of `enumerate` object. This `enumerate` object can then be used directly in for loops or be converted into a list of tuples using `list()` method.

Syntax:

```
enumerate(iterable, start=0)
```

Parameters: Iterable: any object that supports iteration, Start: the index value from which the counter is to be started, by default it is 0

```
In [15]: for i,char in enumerate("hellooo"):
    print(i,char)
```

```
0 h
1 e
2 l
3 l
4 o
5 o
6 o
```

```
In [1]: enumerate("hellooo")
```

```
Out[1]: <enumerate at 0x258296d30b0>
```

```
In [17]: l1 = ["eat","sleep","repeat"]
s1 = "geek"

# creating enumerate objects
obj1 = enumerate(l1)
obj2 = enumerate(s1)

print ("Return type:",type(obj1) )
print (list(enumerate(l1)))

# changing start index to 2 from 0
print (list(enumerate(s1,2)))
```

```
Return type: <class 'enumerate'>
[(0, 'eat'), (1, 'sleep'), (2, 'repeat')]
[(2, 'g'), (3, 'e'), (4, 'e'), (5, 'k')]
```

```
In [18]: for i,char in enumerate([1,2,3]):
    print (i,char) # we get the index along with the item present in the list.
#enumerate is very useful if we want the index counter of the item that we are looping.
```

```
0 1
1 2
2 3
```

```
In [23]: for i,char in enumerate(list(range(0,4))):
    print(i,char)
```

```
0 0
1 1
2 2
3 3
```

```
In [21]: for i,char in enumerate(list(range(0,6))):
    if char == 3:
        print(f'index of 3 is {i}')
```

```
index of 3 is 3
```

```
In [1]: enumerate("hellooo")
```

```
Out[1]: <enumerate at 0x213e539f180>
```

```
In [2]: list(enumerate("hellooo"))
```

```
Out[2]: [(0, 'h'), (1, 'e'), (2, 'l'), (3, 'l'), (4, 'o'), (5, 'o'), (6, 'o')]
```

```
In [3]: for i in enumerate("hellooo"):
    print(i)
```

```
(0, 'h')
(1, 'e')
(2, 'l')
(3, 'l')
(4, 'o')
(5, 'o')
(6, 'o')
```

```
In [4]: for x,y in enumerate ('hello'):
    print(x,y)
```

```
0 h
1 e
2 l
3 l
4 o
```

```
In [5]: for i,k in enumerate('krishna'):
    print(i)
```

```
0
1
2
3
4
5
6

In [6]: for i,char in enumerate(list(range(0,6))):
    if char == 3:
        print(f'index of 3 is {i}')

index of 3 is 3
```

```
In [7]: s1="use string searchings"
s1.index('s')
```

```
Out[7]: 1
```

```
In [8]: s1="use string searchings"
list(enumerate(s1))
```

```
Out[8]: [(0, 'u'),
(1, 's'),
(2, 'e'),
(3, ' '),
(4, 's'),
(5, 't'),
(6, 'r'),
(7, 'i'),
(8, 'n'),
(9, 'g'),
(10, ' '),
(11, 's'),
(12, 'e'),
(13, 'a'),
(14, 'r'),
(15, 'c'),
(16, 'h'),
(17, 'i'),
(18, 'n'),
(19, 'g'),
(20, 's')]
```

Finding index with enumerate:

```
In [2]: s1="use string searchings"
s1.index('s')
```

```
Out[2]: 1
```

```
In [9]: s1="use string searchings"
for x,y in enumerate(s1):
    if y == 's':
        print(f'index of s is {x}')
```

```
index of s is 1
index of s is 4
index of s is 11
index of s is 20
```

Length of String with enumerate:

```
In [10]: st='learning'
len(st)
```

```
Out[10]: 8
```

```
In [11]: l=st.find(st[-1])+1
print(f'length of st is {l}')

length of st is 8
```

```
In [12]: st='go for learninggg'
print(len(st))
```

```
17
```

```
In [13]: st='go for learninggg'
lst=[]
for x,y in enumerate(st):
    if y==st[-1]:
        lst.append(x)
print(lst)
```

```
[0, 14, 15, 16]
```

```
In [14]: [x for x,y in enumerate(st) if y==st[-1]]
```

```
Out[14]: [0, 14, 15, 16]
```

```
In [15]: max([x for x,y in enumerate(st) if y==st[-1]])+1
```

```
Out[15]: 17
```

Length of the string using enumerate:

```
In [16]: st='go for learninggg'
len_st=max([x for x,y in enumerate(st) if y==st[-1]])+1
print(f'length of the string st is {len_st}')
```

```
length of the string st is 17
```

```
In [3]: st='go for learninggg'
for index,char in enumerate(st):
    pass
print(f'length of the string st is {index+1}')
```

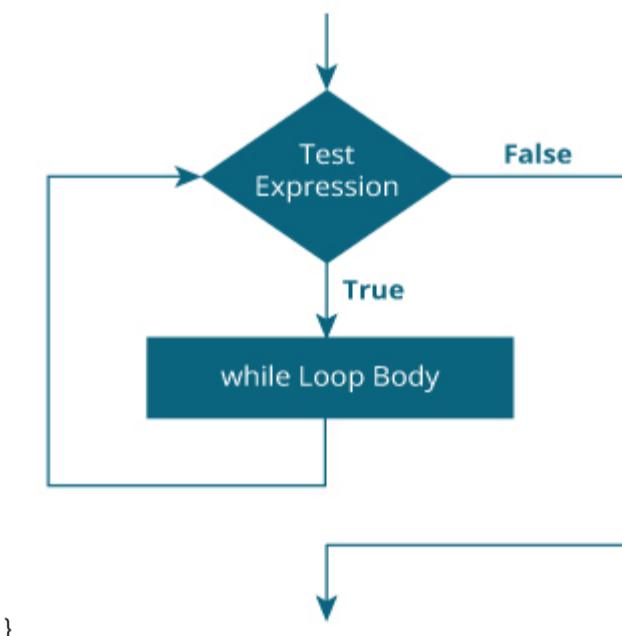
```
length of the string st is 17
```

```
In [4]: st='go for learninggg'
for index,char in enumerate(st, start=1):
    pass
print(f'length of the string st is {index}')
```

```
length of the string st is 17
```

While Loops:

```
while (testExpression)
{
    // statements inside the body of the loop
```



```
In [28]: i=4
while i<5:
    print (i)
    break
```

```
4
```

```
In [30]: i=0
while i<5:
    print (i)
    i=i+1
```

```
0
1
2
3
4
```

- else block will only execute when the while condition is false or after while block execution.

```
In [7]: i=5
while i<3:
    print (i)
    i=i+1
else:
    print("While block failed")
#else block will only execute when the while condition is false or
#after while block execution.
```

```
While block failed
```

```
In [8]: i=0
while i<3:
    print (i)
    i=i+1
else:
    print("While block successfully executed")
```

```
0
1
2
While block successfully executed
```

IMP:

Python supports to have an `else` statement associated with a loop statements.

1. If the `else` statement is used with a `for` loop , the else statement is executed when the loop has exhausted iterating the list.
2. If the `else` statement is used with a `while` loop , the else statement is executed when the condition becomes false or after completion of the loop.

But if there is `break` statement in the `while/for` block then `else` statement won't be executed.

```
In [36]: i=0
while i<3:
    print (i)
    i=i+1
print("While blocked successfully executed")
```

```
0
1
2
While blocked successfully executed
```

```
In [38]: i=0
while i<3:
    print (i)
    i=i+1
    break
else:
#else block will only execute when the while condition is false,
#but when break statement is there with while, else wont be executed.
    print("While blocked successfully executed")
```

```
0
```

```
In [37]: i=0
while i<3:
    print (i)
    i=i+1
    break
print("While blocked successfully executed")
```

```
0
While blocked successfully executed
```

```
In [42]: for i in [1,2,3]:
    print(i)
else:
    print('x')
```

```
1
2
3
x
```

```
In [43]: for i in [1,2,3]:
    print(i)
    break
else:
    print('x')
```

```
1
```

```
In [44]: for i in [1,2,3]:
    print(i)
    break
print('x')
```

```
1
x
```

```
In [45]: my_list=[1,2,3]
for i in my_list:
    print (i)
```

```
1
2
3
```

```
In [1]: my_list=[1,2,3]
i=0
while i<len(my_list):
    print(my_list[i])
    i=i+1
```

```
1
2
3
```

```
In [ ]: while True:
    input("Say Something: ")
```

```
Say Something: hi
Say Something: hi
```

```
In [3]: while True:
    response=input("Say Something: ")
    if (response=='bye'):
        break
```

```
Say Something: hi
Say Something: hi
Say Something: bye
```

break,continue,pass:

When we use the `break` segment it breaks out and closes loop. So when we used a break statement we broke out of this loop. The break statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C. The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

When the interpreter executes the `continue`, the interpreter immediately goes to top of the loop, leaving all the code block below the continue statement. The continue statement in Python returns the control to the beginning of the while loop. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The continue statement can be used in both while and for loops.

The `pass` statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. The pass statement is a null operation; nothing happens when it executes. The pass is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

```
In [7]: for i in [1,2,3]:
    print(i)

i=0
while i<3:
    print (i)
    i=i+1
```

```
1
2
3
0
1
2
```

```
In [8]: for i in [1,2,3]:
    print(i)      #Loop doesn't iterate
    break        #breaks the Loops and closes the for Loop

i=0
while i<3:
    print (i)
    i=i+1        #Loop doesn't iterate
    break        #breaks the Loops and closes the while Loop
```

```
1
0
```

```
In [3]: for i in [1,2,3]: #here Loop iterates but the print statement doesn't execute
    continue
    print(i)
```

```
In [4]: i=0
while i<3:      #here Loop iterate
    continue
    print (i)
    i=i+1
#here i values never increments hence i=0 and i will always be less than 3.
```

```
-----
KeyboardInterrupt                                Traceback (most recent call last)
Cell In[4], line 3
  1 i=0
  2 while i<3:      #here loop iterate
----> 3     continue
  4     print (i)
  5     i=i+1
```

KeyboardInterrupt:

```
In [5]: for i in [1,2,3]: #here Loop iterate
          #if the statement here is left
i=0
while i<3:      #here loop iterate
    print (i)
    i=i+1
```

```
File "<ipython-input-5-c721dbcaa781>", line 3
```

```
    i=0
    ^
IndentationError: expected an indented block
```

```
In [12]: for i in [1,2,3]:
# if the statement here is not written ,error occurs.
#so to overcome this situation we use pass.
#in general we write a loop and if we dont know what to write in the block
#we can use pass
#and execute for now, when we get idea we can write that code in the loop.
```

```
File "<ipython-input-12-e6386936a847>", line 3
```

```
#      n general we write a loop and if we dont know what to write in the block we can use pass and execute for now, when we
get idea we can write that code in the loop.
```

```
^
```

SyntaxError: unexpected EOF while parsing

```
In [2]: for i in [1,2,3]:
        pass

i=0
while i<3:
    print (i)
    i=i+1
```

```
0
```

```
1
```

```
2
```

```
In [2]: for letter in 'Python':      # First Example
    if letter == 'h':
        break
    print('Current Letter :', letter)

var = 10                      # Second Example
while var > 0:
    print ('Current variable value :', var)
    var = var -1
    if var == 5:
        break
```

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
```

```
In [19]: for letter in 'Python':      # First Example
    if letter == 'h':
        continue
    print('Current Letter :', letter)

var = 5                      # Second Example
while var > 0:
    var = var -1
    if var == 3:
        continue
    print ('Current variable value :', var)
```

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 4
Current variable value : 2
Current variable value : 1
Current variable value : 0
```

```
In [2]: for letter in 'Python':
    if letter == 'h':
        pass
    print ('This is pass block')
    print ('Current Letter :', letter)

Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
```

The preceding code does not execute any statement or code if the value of letter is 'h'. The pass statement is helpful when you have created a code block but it is no longer required. You can then remove the statements inside the block but let the block remain with a pass statement So that it doesn't interfere with other parts of the code.

GUI:

```
In [3]: #Exercise!
#Display the image where the 0 is going to be ' ', and the 1 is going to be '*'.
#This will reveal an image!
picture = [
    [0,0,0,1,0,0,0],
    [0,0,1,1,1,0,0],
    [0,1,1,1,1,1,0],
    [1,1,1,1,1,1,1],
    [0,0,0,1,0,0,0],
    [0,0,0,1,0,0,0]
]

for row in picture:
    for pixel in row:
        if (pixel==1):
            print('*')
        else:
            print(' ')
```

*

*

*

*

*

*

*

*

*

*

*

*

*

*

*

*

*

```
In [43]: picture = [
    [0,0,0,1,0,0,0],
    [0,0,1,1,1,0,0],
    [0,1,1,1,1,1,0],
    [1,1,1,1,1,1,1],
    [0,0,0,1,0,0,0],
    [0,0,0,1,0,0,0]
]
```

```

for row in picture:
    for pixel in row:
        if (pixel==1):
            print('*', end = "")
        else:
            print(' ', end = "")

*     ***     ***** ****   *   *

```

```

In [4]: picture = [
    [0,0,0,1,0,0,0],
    [0,0,1,1,1,0,0],
    [0,1,1,1,1,1,0],
    [1,1,1,1,1,1,1],
    [0,0,0,1,0,0,0],
    [0,0,0,1,0,0,0]
]
for row in picture:
    for pixel in row:
        if (pixel):
            print('*', end = "")
        else:
            print(' ', end = "")
    print('#we need to write this print statement as we need print the next row in new line.
#because print() bydefault prints in new line, so its enough to just write print('') to get a new line

*
***  

*****  

*****  

*  

*
```

```

In [7]: picture = [
    [0,0,0,1,0,0,0],
    [0,0,1,1,1,0,0],
    [0,1,1,1,1,1,0],
    [1,1,1,1,1,1,1],
    [0,0,0,1,0,0,0],
    [0,0,0,1,0,0,0]
]

for row in picture:
    for pixel in row:
        if (pixel==1):
            print('*', end="")
        else:
            print(' ', end="") #space in the print statement also matters
    print()

*
***  

*****  

*****  

*  

*
```

Python end parameter in print():

Python's `print()` function comes with a parameter called `end`. By default, the value of this parameter is `\n`, i.e. the new line character. You can end a print statement with any character/string using this parameter.

By default python's `print()` function ends with a newline. A programmer with C/C++ background may wonder how to print without newline.

<https://www.geeksforgeeks.org/gfact-50-python-end-parameter-in-print/>

<https://www.studytonight.com/post/the-sep-and-end-parameters-in-python-print-statement>

```
In [8]: print("Welcome to")
print("GeeksforGeeks")
```

Welcome to
GeeksforGeeks

```
In [24]: # ends the output with a <space>
print("Welcome to" , end = ' ')
print("GeeksforGeeks", end = ' ')
```

Welcome to GeeksforGeeks

Exercise:

```
In [61]: #Check for duplicates in the list:
some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']
duplicates = []
for value in some_list:
    if some_list.count(value) > 1:
        duplicates.append(value)

print(duplicates)
```

```
['b', 'b', 'n', 'n']
```

```
In [44]: #Check for duplicates in the list:
some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']

duplicates = []
for value in some_list:
    if some_list.count(value) > 1:
        if value not in duplicates:
            duplicates.append(value)

print(duplicates)
```

```
['b', 'n']
```

```
In [51]: some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']
x=len(some_list)
for i in range(x):
    k = i + 1
    for j in range(k, x):
        if some_list[i] == some_list[j]:
            print(some_list[i])
```

```
b
```

```
n
```

```
In [59]: repeated=[]
some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']
x=len(some_list)
for i in range(x):
    k = i + 1
    for j in range(k, x):
        if some_list[i] == some_list[j]:
            repeated.append(some_list[i])
print(repeated)
```

```
['b', 'n']
```

```
In [60]: repeated=[]
some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']
x=len(some_list)
for i in range(x):
    k = i + 1
    for j in range(k, x):
        if some_list[i] == some_list[j] and some_list[i] not in repeated :
            repeated.append(some_list[i])
print(repeated)
```

```
['b', 'n']
```

Functions:

```
In [62]: def python():
    print("Programming language")

python()
```

```
Programming language
```

```
In [63]: def duplicates():
    repeated=[]
    some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']
    x=len(some_list)
    for i in range(x):
        k = i + 1
        for j in range(k, x):
            if some_list[i] == some_list[j] and some_list[i] not in repeated :
                repeated.append(some_list[i])
    print(repeated)
```

```
In [64]: duplicates()
```

```
['b', 'n']
```

```
In [67]: print(duplicates) # gives the location where the function is stored.

<function duplicates at 0x000000C6FE1D7558>
```

Call by Object Reference in Python

In Python, we have `call by object reference` not call by value or call in reference

<https://medium.com/@lokeshsharma596/is-python-call-by-value-or-call-by-reference-2dd7db74dbd0>

Python uses a mechanism that is often called "call by object reference" or "call by sharing." This means that when you pass an argument to a function in Python, you are actually passing a reference to the object, not the object itself. However, this behavior can sometimes resemble both call by reference and call by value, depending on how you look at it.

In Python:

- Immutable objects like integers, floats, strings, and tuples are essentially passed by value. Changes made to these objects within a function do not affect the original objects outside the function.
- Mutable objects like lists and dictionaries are passed by reference. Changes made to these objects within a function can affect the original objects outside the function.

Let's illustrate this with examples:

Passing an integer (immutable):

```
def modify_int(x):
    x += 1

num = 10
modify_int(num)
print(num) # Output: 10
```

Here, the integer `num` remains unchanged because integers are immutable.

Passing a list (mutable):

```
def modify_list(lst):
    lst.append(4)

my_list = [1, 2, 3]
modify_list(my_list)
print(my_list) # Output: [1, 2, 3, 4]
```

In this case, the list is modified within the function and the changes are reflected outside as well.

So, while Python's behavior can be described as "call by object reference," the distinction between call by reference and call by value becomes less clear due to the different behaviors of mutable and immutable objects.

```
In [5]: def modify_int(x):
    x += 1

num = 10
print(f'before function call: {num}')
modify_int(num)
print(f'after function call: {num}')
print(num) # Output: 10
```

before function call: 10
after function call: 10
10

```
In [6]: def modify_list(lst):
    lst.append(4)

my_list = [1, 2, 3]
print(f'before function call: {my_list}')
modify_list(my_list)
print(f'before function call: {my_list}')
print(my_list) # Output: [1, 2, 3, 4]
```

before function call: [1, 2, 3]
before function call: [1, 2, 3, 4]
[1, 2, 3, 4]

```
In [32]: def val(x):
    x=15
    print(x,id(x))
x=10
val(x)
print(x,id(x))
```

15 2284391629488
10 2284391629328

```
In [33]: lst=[1,2,3]
def val(lst):
    lst.append(4)
    print(lst,id(lst))
lst=[1,2,3]
print(lst,id(lst))
val(lst)
```

[1, 2, 3] 2284502668096
[1, 2, 3, 4] 2284502668096

```
In [1]: lst=[1,2,3]
def val(lst):
    lst.append(4)
    print(lst,id(lst))
lst=[1,2,3]
val(lst)
print(lst,id(lst))
```

```
[1, 2, 3, 4] 2797341937792
[1, 2, 3, 4] 2797341937792
```

```
In [34]: def modify_list(my_list):
    my_list.append(4)
```

```
my_list = [1, 2, 3]
modify_list(my_list)
print(my_list) # Output: [1, 2, 3, 4]
```

```
[1, 2, 3, 4]
```

```
In [35]: def mlist(lst):
    lst.append(4)
lst=[1,2,3]
mlist(lst)
print(lst)
```

```
[1, 2, 3, 4]
```

```
In [36]: def mlist(lst):
    lst.append(4)
    print(lst)
lst=[1,2,3]
mlist(lst)
print(lst)
```

```
[1, 2, 3, 4]
```

```
[1, 2, 3, 4]
```

```
In [37]: def modify_int(x):
    x += 1
    print(x)
num = 10
modify_int(num)
print(num) # Output: 10
```

```
11
```

```
10
```

```
In [2]: def modify_int(x):
    x += 1
    print(x)
num = 10
print(num) # Output: 10
modify_int(num)
```

```
10
```

```
11
```

```
In [38]: #Mutable objects
def mlist(lst):
    lst.append(4)
    print(lst,id(lst))
lst=[1,2,3]
mlist(lst)
print(lst,id(lst))
```

```
#immutable objects
```

```
def modify_int(num):
    num += 1
    print(num,id(num))
num = 10
modify_int(num)
print(num,id(num)) # Output: 10
```

```
[1, 2, 3, 4] 2284502635264
```

```
[1, 2, 3, 4] 2284502635264
```

```
11 2284391629360
```

```
10 2284391629328
```

Parameters and Arguments(Call and Invoke) :

- Parameters are used when we define the function and arguments are used when we call the function defined

- Arguments are used as the actual values we provide to a function,

arguments are the actual value we provide to the function and the name of the variables

that we use that we receive are called parameters.

When a function is called, the values that are passed during the call are called as arguments.

The values which are defined at the time of the function prototype or definition of the function are called as parameters.

- During the time of call, each argument is always assigned to the parameter in the function definition.

Positional Arguments:

```
In [1]: #parameters
def coding(language,version):    # here language and version are parameters
    print(f'{language} - version {version} is really cool')

#arguments
coding('python',3.7) #python and 3.7 are arguments
coding('conda',2.4)

python - version 3.7 is really cool
conda - version 2.4 is really cool
```

```
In [2]: def coding(language,version):
    print(f'{language} - version {version} is really cool')

x=input("Enter language: ")
y=input("Enter version: ")
coding(x,y)

Enter language: Python
Enter version: 3.7
Python - version 3.7 is really cool
```

positional arguments are arguments that require to be in the proper position.

```
In [3]: #parameters
def coding(language,version):    # here language and version are parameters
    print(f'{language} - version {version} is really cool')

#(positional)arguments
coding(3.7,'python') #python and 3.7 are arguments

#this output is not correct as the language and version are changed. Reason is we have changed the arguments position.

3.7 - version python is really cool
```

Keyword Arguments:

```
In [26]: def coding(language,version):
    print(f'{language} - version {version} is really cool')

#Keyword Arguments
coding(version=3.7,language='python') #independent of position of the arguments
#now we will get the output as required even if the arguments position is changed because we used keyword arguments.

python - version 3.7 is really cool
```

Default Parameters:

```
In [18]: #Default Parameters
def coding(language='python',version=3.7):
    print(f'{language} - version {version} is really cool')

coding('conda',2.4)
coding(language='Robo',version=2.0)
coding(version=1.0,language='pandas')
coding() # if we fail to give arguments, then the default parameters will be taken as arguments

conda - version 2.4 is really cool
Robo - version 2.0 is really cool
pandas - version 1.0 is really cool
python - version 3.7 is really cool
```

```
In [3]: #Default Parameters
def coding(language='python',version=3.7):
    print(f'{language} - version {version} is really cool')
coding()
coding('conda')
coding(version=2.6)

python - version 3.7 is really cool
conda - version 3.7 is really cool
python - version 2.6 is really cool
```

IMP:

```
In [3]: #parameters
def coding(language,version):
    pass

#Default Parameters
def coding(language='python',version=3.7):
    print(f'{language} - version {version} is really cool')

#positional arguments
coding('conda',2.4)
```

```
#Keyword Arguments
coding(version=2.0,language='Robo')

coding()

conda - version 2.4 is really cool
Robo - version 2.0 is really cool
python - version 3.7 is really cool
```

Type Hinting:

- Type hinting in Python is a way to provide additional information about the types of variables, function parameters, and return values in your code.

It doesn't affect the actual execution of the code but serves as a form of documentation that helps developers understand the expected types and improves code readability. Type hints can also be used by various tools and linters to catch type-related errors and provide better code analysis.

Type hints were introduced in Python 3.5 and became more widely adopted with the release of Python 3.6 and later versions.

Here's a simple example to illustrate type hinting:

```
def add_numbers(a: int, b: int) -> int:
    return a + b

result = add_numbers(5, 10)
print(result)
```

In this example, the `add_numbers` function takes two parameters, `a` and `b`, both of which are annotated with the type `int`. The `-> int` after the parameter list indicates that the function is expected to return an integer. These type annotations provide clear information about the expected types of the function's arguments and its return value.

Type hints are not enforced by the Python interpreter itself, which means you can still run code with incorrect types. However, using tools like `mypy`, which is a popular static type checker for Python, you can analyze your code for type-related issues before running it. If there's a mismatch between the type hints and the actual code, `mypy` would raise an error.

For instance, consider this incorrect usage:

```
result = add_numbers("5", 10) # This will raise a type hinting error with mypy
```

- Type hinting can be applied to variables, function parameters, function return values, and even more complex data structures like lists, dictionaries, and classes. It helps improve code quality, makes the codebase more maintainable, and reduces the chances of runtime errors related to type mismatches.

```
In [19]: def cod(lan:str, ver:float) -> str:
    print(f'{lan} of version {ver} is cool')
cod('python',3.7)

python of version 3.7 is cool
```

```
In [20]: def cod(lan:str, ver:float) -> str:
    print(f'{lan} of version {ver} is cool')
cod(2,'krish') #type hinting doesn't give any error it is just for documentation purpose

2 of version krish is cool
```

Type hinting in keyword args:

```
In [21]: def cod(lan:str, ver:float) -> str:
    print(f'{lan} of version {ver} is cool')
cod(ver=3.6, lan='JAVA')

JAVA of version 3.6 is cool
```

Type hinting with Default params:

```
In [22]: def coding(lan:str='python', version:float=3.7)->str:      #parameter:datatype=defaultvalue
    print(f'{lan} - version {version} is cool')
coding()

python - version 3.7 is cool
```

```
In [23]: def coding(lan:str='python', version:float=3.7)->str:
    print(f'{lan} - version {version} is really cool')
coding('conda',3.6)
coding(version=3.7, lan='scala')

conda - version 3.6 is really cool
scala - version 3.7 is really cool
```

```
In [5]: # Explain the following piece of code:

...
def init(
    self,
    date: date = date.today(),
    url: str | None = None,
    id: str | None = None,
```

```

        type: None | Literal['file', 'folders'] = 'file',
        exists_ok: bool = False
    ) -> None:
...

```

Out[5]:

```
"\nndef init(\n    self,\n    date: date = date.today(),\n    url: str | None = None,\n    id: str | None = Non
e,\n    type: None | Literal['file', 'folders'] = 'file',\n    exists_ok: bool = False\n) -> None:\n"
```

Certainly! The provided piece of code defines an `init` method within a Python class. Let's break down the code step by step to understand its functionality:

1. `def init(self, ...)` -> `None`: This line defines a constructor method named `init` within a class. The `self` parameter refers to the instance of the class that is being created. The `...` indicates that there are additional parameters following.
2. `date: date = date.today()`: This parameter is named `date` and has a default value of the current date obtained using the `date.today()` method. The parameter is expected to be of the `date` type.
3. `url: str | None = None`: This parameter is named `url` and has a default value of `None`. It can hold a value that is either a string (`str`) or `None`.
4. `id: str | None = None`: This parameter is named `id` and also has a default value of `None`. Similar to the `url` parameter, it can hold a value that is either a string (`str`) or `None`.
5. `type: None | Literal['file', 'folders'] = 'file'`: This parameter is named `type` and has a default value of `'file'`. It can hold a value that is either `None` or one of the two string literals: `'file'` or `'folders'`. The `Literal` type hint is used to specify the exact allowed string literal values.

Literal types let you indicate that an expression is equal to some specific primitive value. For example, if we annotate a variable with type `Literal["foo"]`, mypy will understand that variable is not only of type `str`, but is also equal to specifically the string "foo".

To use `Literal`, we simply parametrize it with the allowed literal values:

```

from typing import Literal

game: Literal["checkers", "chess"]

```

We've declared that the `game` variable has only two possible values: the strings `"checkers"` and `"chess"`.

6. `exists_ok: bool = False`: This parameter is named `exists_ok` and has a default value of `False`. It's a boolean parameter that indicates whether the operation is allowed to proceed if the specified resource already exists.
7. `-> None`: This part of the method definition indicates that the method returns `None`, meaning it doesn't return any value.

In summary, the `init` method is intended to initialize an instance of the class. It takes several parameters:

- `date`: A date that defaults to the current date.
- `url`: A URL represented as a string, or `None`.
- `id`: An ID represented as a string, or `None`.
- `type`: A type that can be `None`, `'file'`, or `'folders'`.
- `exists_ok`: A boolean indicating whether the operation can proceed if the resource already exists.

These parameters provide various options for initializing the instance with specific values. The method doesn't return anything (`None`). This type of method is commonly used in Python classes to set up initial attributes and state when creating new instances of the class.

In [7]:

```

from typing import List, Tuple

def process_data(data: List[int]) -> Tuple[int, int]:
    return sum(data), len(data)

data = [1, 2, 3, 4, 5]
print(process_data(data)) # Output: (15, 5)

```

(15, 5)

Here, `List[int]` and `Tuple[int, int]` are type hints indicating that "data" is a list of integers and the function returns a tuple of two integers.

Return:

Well functions are just like that they always have to return something and when they don't return anything like there's no `return` keyword, they automatically return `None`.

```
In [7]: def sum(num1, num2):
    num1+num2
print(sum(4,5))
```

None

```
In [8]: def sum(num1, num2):
    return num1+num2
print(sum(4,5))
```

9

```
In [9]: def sum(num1, num2):
    return num1+num2
sum(4,5)
```

Out[9]: 9

if we add `return` keyword in the function it's going to say, as soon as the interpreter comes to return statement, it asks the interpreter to exit this function and

when you exit this function, I want you to return whatever this expression gives us.

```
In [34]: #A function either modifies something in our program or returns something
def sum(num1, num2):
    print("hi") #here the functions printed 'hi' i.e., modified something in the program but didnt return anything.
    num1+num2
print(sum(4,5))
```

hi
None

```
In [37]: def sum(num1, num2):
    return num1+num2
total=sum(4,5)
print(sum(total,11))
print(sum(21,sum(3,6)))
```

20
30

```
In [10]: def sum(num1,num2):
    def sum2(num1,num2):
        return num1+num2
    return sum2

sum(10,5)
```

Out[10]: <function __main__.sum.<locals>.sum2(num1, num2)>

```
In [11]: def sum(num1,num2):
    def sum2(num1,num2):
        return num1+num2
    return sum2()

sum(1,6)
```

TypeError Traceback (most recent call last)
Cell In[11], line 6
 3 return num1+num2
 4 return sum2()
----> 6 sum(1,6)

Cell In[11], line 4, in sum(num1, num2)
 2 def sum2(num1,num2):
 3 return num1+num2
----> 4 return sum2()

TypeError: sum.<locals>.sum2() missing 2 required positional arguments: 'num1' and 'num2'

```
In [12]: def sum(num1,num2):
    def sum2(num1,num2):
        return num1+num2
    return sum2(num1,num2)

sum(1,6)
```

Out[12]: 7

```
In [14]: def sum(num1,num2):
    def sum2(n1,n2):
        return n1+n2
    return sum2(num1,num2)

sum(10,5)
```

Out[14]: 15

In []:

```
In [15]: def sum(num1,num2):
    def sum2(num1,num2):
        return num1+num2
    return sum2

total=sum(10,5)
print(total)
# If I click 'Run All', I get this function now.
#So total is going to equal this function that we returned. And it's in memory.

<function sum.<locals>.sum2 at 0x0000028B4E908160>
```

The above code defines a function sum that takes two arguments num1 and num2. Inside this function, another nested function sum2 is defined that takes the same arguments and returns their sum. Then, the sum function returns the sum2 function itself rather than calling it. When sum(10, 5) is called, it returns the sum2 function. So, total now holds a reference to the sum2 function.

```
total = sum2
```

When total is printed, it will display the memory address or representation of the sum2 function object, not the result of the addition of 10 and 5. If you want to get the result of the addition, you should call the returned function total with the appropriate arguments

```
In [22]: def sum(num1,num2):
    def sum2(num1,num2):
        return num1+num2
    return sum2

total=sum(10,5)
print(total(1,5)) #total(1,5)=sum2(1,5)
```

```
6
```

```
In [28]: def sum(num1,num2):
    def sum2(n1,n2):
        return num1+num2
    return sum2
total=sum(10,5)
print(total(1,5))
```

```
15
```

```
In [ ]:
```

```
In [11]: def sum(num1,num2):
    def sum2(num1,num2):
        return num1+num2
    return sum2(num1,num2)

total=sum(10,5)
print(total)
```

```
15
```

```
In [15]: def sum(num1,num2):
    def sum2(n1,n2):
        return n1+n2
    return sum2(num1,num2)

total=sum(10,5)
print(total)
```

```
15
```

```
In [24]: def sum(num1,num2):
    def sum2(n1,n2):
        return n1+n2
    return sum2(num1,num2)

total=sum(10,5)
print(total(4,6))
```

```
-----  
TypeError                                     Traceback (most recent call last)  
Cell In[24], line 7  
      4     return sum2(num1,num2)  
      6 total=sum(10,5)  
----> 7 print(total(4,6))  
  
TypeError: 'int' object is not callable
```

```
In [13]: def sum(num1,num2):
    def sum2(n1,n2):
        return n1+n2
    return sum2(num1,num2)
    print("Hello")

total=sum(10,5)
print(total)
```

```
15
```

return keyword automatically exits the function. so that in here if I added another

piece of code like print Hello, its still valid but if I run this I still get 15, because the interpreter never gets to line five because as soon as we return something from a function it exits that function.

```
In [24]: def sum(num1, num2):
    print(num1+num2)
total=sum(4,5)
print(sum(total,11))
print(sum(21,sum(3,6)))
```

9

```
-----
TypeError                                         Traceback (most recent call last)
Cell In[24], line 4
      2     print(num1+num2)
      3 total=sum(4,5)
----> 4 print(sum(total,11))
      5 print(sum(21,sum(3,6)))

Cell In[24], line 2, in sum(num1, num2)
      1 def sum(num1, num2):
----> 2     print(num1+num2)

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

```
In [26]: def sum(num1, num2):
    return(num1+num2)
total=sum(4,5)
print(total)
print(sum(total,11))
print(sum(21,sum(3,6)))
```

```
9
20
30
```

In []:

```
In [25]: def fn(a,b):
    def fn2(c,d):
        return a+b
    return fn2
var=fn(3,4)
var #var=fn2
```

Out[25]: <function __main__.fn.<locals>.fn2(c, d)>

```
In [26]: def fn(a,b):
    def fn2(c,d):
        return a+b
    return fn2
var=fn(3,4)
var()
```

```
-----
TypeError                                         Traceback (most recent call last)
Cell In[26], line 6
      4     return fn2
      5 var=fn(3,4)
----> 6 var()

TypeError: fn.<locals>.fn2() missing 2 required positional arguments: 'c' and 'd'
```

```
In [27]: def fn(a,b):
    def fn2(c,d):
        return a+b
    return fn2
var=fn(3,4)
var(10,15)
```

Out[27]: 7

```
In [29]: def fn(num1,num2):
    def fn2(num1,num2):
        return num1+num2
    return fn2(num1,num2)

total=fn(10,5)
print(total)
```

15

```
In [30]: def fn(num1,num2):
    def fn2(n1,n2):
        return n1+n2
    return fn2(num1,num2)
```

```
total=fn(10,5)
print(total)
```

15

Exercise: Tesla

```
In [ ]: age = input("What is your age?: ")

if int(age) < 18:
    print("Sorry, you are too young to drive this car. Powering off")
elif int(age) > 18:
    print("Powering On. Enjoy the ride!");
elif int(age) == 18:
    print("Congratulations on your first year of driving. Enjoy the ride!")

#1. Wrap the above code in a function called checkDriverAge().
#Whenever you call this function, you will get prompted for age.
# Notice the benefit in having checkDriverAge() instead of copying and pasting the function everytime?

#2 Instead of using the input(). Now, make the checkDriverAge() function accept an argument of age, so that if you enter:
#checkDriverAge();
#it returns "Powering On. Enjoy the ride!"
#also make it so that the default age is set to 0 if no argument is given.
```

```
In [19]: #1. Wrap the above code in a function called checkDriverAge(). Whenever you call this function,
#you will get prompted for age.
# Notice the benefit in having checkDriverAge() instead of copying and pasting the function everytime?
def checkDriverAge():
    age = input("What is your age?: ")
    if int(age) < 18:
        print("Sorry, you are too young to drive this car. Powering off")
    elif int(age) > 18:
        print("Powering On. Enjoy the ride!");
    elif int(age) == 18:
        print("Congratulations on your first year of driving. Enjoy the ride!")

checkDriverAge()
checkDriverAge()
checkDriverAge()
```

```
What is your age?: 17
Sorry, you are too young to drive this car. Powering off
What is your age?: 19
Powering On. Enjoy the ride!
What is your age?: 18
Congratulations on your first year of driving. Enjoy the ride!
```

```
In [1]: #2 Instead of using the input(). Now, make the checkDriverAge() function accept an argument of age, so that if you enter:
#checkDriverAge(92);
#it returns "Powering On. Enjoy the ride!"
#also make it so that the default age is set to 0 if no argument is given.

def checkDriverAge(age=0):
    if int(age) < 18:
        print("Sorry, you are too young to drive this car. Powering off")
    elif int(age) > 18:
        print("Powering On. Enjoy the ride!");
    elif int(age) == 18:
        print("Congratulations on your first year of driving. Enjoy the ride!")

checkDriverAge(17)
checkDriverAge(92)
checkDriverAge(18)
checkDriverAge() #here the function takes default parameter age=0 as argument
```

```
Sorry, you are too young to drive this car. Powering off
Powering On. Enjoy the ride!
Congratulations on your first year of driving. Enjoy the ride!
Sorry, you are too young to drive this car. Powering off
```

Docstrings:

<https://www.geeksforgeeks.org/python-docstrings/#:~:text=Learning%20in%202020-,Python%20Docstrings,a%20specific%20segment%20of%20code.>

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods.

Python docstrings are strings used right after the definition of a function, method, class, or module. They are used to document our code. We can access these docstrings using the **doc** attribute.

The doc string line should begin with a capital letter and end with a period. The first line should be a short description. If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

- **Declaring Docstrings :** The docstrings are declared using """"triple double quotes"""" just below the class, method or function declaration. All functions should have a docstring.

- **Accessing Docstrings**: The docstrings can be accessed using the `__doc__` method of the object or using the help function.

```
In [12]: def test(a):
    ...
    Info:This function just tests and prints parameter a
    ...
    print(a)
test('hi')
test.__doc__
```

hi
Out[12]: '\n Info:This function just tests and prints paameter a\n '

```
In [1]: def test(a):
    ...
    Info:This function just tests and prints parameter a
    ...
    print(a)

help(test)
```

Help on function test in module __main__:

```
test(a)
Info:This function just tests and prints parameter a
```

```
In [4]: def my_function():
    """Demonstrates docstrings and does nothing really."""
    return None

print ("Using __doc__:")
print (my_function.__doc__)

print("*****")
print ("Using help:")
help(my_function)
```

Using `__doc__`:

Demonstrates docstrings and does nothing really.

Using help:

Help on function my_function in module __main__:

```
my_function()
    Demonstrates docstrings and does nothing really.
```

Clean code:

```
In [16]: def is_even(num):
    if num%2==0:
        return True
    else:
        return False
is_even(21)
```

Out[16]: False

```
In [17]: def is_even(num):
    if num%2==0:
        return True
    return False
is_even(21)
```

Out[17]: False

```
In [22]: def is_even(num):
    return num%2==0
is_even(20)
```

Out[22]: True

*args and *kwargs:

arguments and keyword arguments: <https://www.geeksforgeeks.org/args-kwargs-python/>

<https://www.tutorialspoint.com/args-and-kwargs-in-python>

<https://www.geeksforgeeks.org/packing-and-unpacking-arguments-in-python/>

Note: We use the "wildcard" or "`args OR **kwargs`" notation like this – `args OR **kwargs` – as our function's argument when we are not sure about the number of arguments we will pass in a function.

Rule for writing parameters while defining the function: `params, *args, default params, **kwargs`

```
In [9]: #Rule:params, *args, default params, **kwargs
def super_fun(a):
    return sum(a)

super_fun(1,2,3)
```

```
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-9-1086e82f2688> in <module>
      3     return sum(a)
      4
----> 5 super_fun(1,2,3)

TypeError: super_fun() takes 1 positional argument but 3 were given
```

```
In [27]: def super_fun(*a):
    print(a)
    print(*a)
    return sum(a)  #sum is a built-in function in python, so there is no need to define its operation again

super_fun(1,2,3)

(1, 2, 3)
1 2 3
6
```

```
Out[27]:
```

```
In [28]: #Here *args is parameter for super_fun()
def super_fun(*args):
    print(args)  #gives tuple of args
    print(*args)  #gives the args
    return sum(args)  #sum is a built-in function in python, so there is no need to define its operation again

super_fun(1,2,3)

(1, 2, 3)
1 2 3
6
```

```
Out[28]:
```

```
In [31]: def super_fun(*args,**kwargs):
    return sum(args)

super_fun(1,2,3, num1=4, num2=5)
```

```
Out[31]: 6
```

```
In [32]: #Here *args,**kwargs are parameters for super_fun()
def super_fun(*args,**kwargs):
    print(kwargs)  #gives the dictionary of kwargs
    return sum(args)

super_fun(1,2,3, num1=4, num2=5)

{'num1': 4, 'num2': 5}
```

```
Out[32]: 6
```

```
In [34]: def super_fun(*args,**kwargs):
    return sum(args)+sum(kwargs.values())
#as kwargs gives dictionary , we can grab the values off dictionary but using '.values()'

super_fun(1,2,3, num1=4, num2=5)
```

```
Out[34]: 15
```

```
In [7]: def super_fun(*args,**kwargs):
    total=0
    for item in kwargs.values():
        total=total+item
    print (total)
    return sum(args)+total

super_fun(1,2,3, num1=4, num2=5)
```

```
9
15
```

We have the `*args` which allow us to grab these positional arguments and just sum everything. And we also have `**kwargs` which allow us to grab any number of keyword arguments and get a dictionary which comes as `**kwargs` and then use them however we want in our case we're looping over all the values so items in `**kwargs` values and then I'm just going to total all those items have a total and just return the sum.

```
In [35]: def findproduct(*many_nums):
    result = 1
    for num in many_nums:
        result = result * num
    print("Multiplication result:",result)
```

```
findproduct(3,9)
findproduct(2,11,10)
```

Multiplication result: 27
Multiplication result: 220

```
In [27]: def findproduct(*args):
    result = 1
    for num in args:
        result = result * num
    print("Multiplication result:",result)

findproduct(3,9)
findproduct(2,11,10)
```

Multiplication result: 27
Multiplication result: 220

```
In [3]: def country_details(**state_info):
    print('\n')
    for k,v in state_info.items():
        print("{} is {}".format(k,v))

country_details(StateName="Telangana", Capital="Hyderabad", Population=3400000)
country_details(StateName="Andhra Pradesh", Capital="Amaravati", Population=1000000, ForestCoverage="30%")
```

StateName is Telangana
Capital is Hyderabad
Population is 3400000

StateName is Andhra Pradesh
Capital is Amaravati
Population is 1000000
ForestCoverage is 30%

```
In [5]: def country_details(**state_info):
    for k,v in state_info.items():
        print("{} is {}".format(k,v))

country_details(StateName="Telangana", Capital="Hyderabad", Population=3400000)
country_details(StateName="Andhra Pradesh", Capital="Amaravati", Population=1000000, ForestCoverage="30%")
```

StateName is Telangana
Capital is Hyderabad
Population is 3400000
StateName is Andhra Pradesh
Capital is Amaravati
Population is 1000000
ForestCoverage is 30%

```
In [6]: def country_details(**kwargs):
    print('')
    for k,v in kwargs.items():
        print(f"{k} is {v}")

country_details(StateName="Telangana", Capital="Hyderabad", Population=3400000)
country_details(StateName="Andhra Pradesh", Capital="Amaravati", Population=1000000, ForestCoverage="30%")
```

StateName is Telangana
Capital is Hyderabad
Population is 3400000

StateName is Andhra Pradesh
Capital is Amaravati
Population is 1000000
ForestCoverage is 30%

```
In [13]: def country_details(**kwargs):
    print(kwargs) #gives the dictionary
    print(kwargs.items())
    print() #for an empty line

country_details(StateName="Telangana", Capital="Hyderabad", Population=3400000)
country_details(StateName="Andhra Pradesh", Capital="Amaravati", Population=1000000, ForestCoverage="30%")

{'StateName': 'Telangana', 'Capital': 'Hyderabad', 'Population': 3400000}
dict_items([('StateName', 'Telangana'), ('Capital', 'Hyderabad'), ('Population', 3400000)])

{'StateName': 'Andhra Pradesh', 'Capital': 'Amaravati', 'Population': 1000000, 'ForestCoverage': '30%'}
dict_items([('StateName', 'Andhra Pradesh'), ('Capital', 'Amaravati'), ('Population', 1000000), ('ForestCoverage', '30%')])
```

IMP:

Parameters defining:

```
In [3]: #Rule:params, *args, default params, **kwargs
def super_fun(name, *args, greet='Hi', **kwargs):
    total=0
    for item in kwargs.values():
        total=total+item
```

```

print(f'hey, {greet} {name}.')
return sum(args)+total

```

hey, Hi Coder.

Out[3]:

```

In [14]: def testfn(name,*args,greet='hello',**kwargs):
    total=0
    for i in kwargs.values():
        total+=i
    return (f'{greet} {name}, value is {sum(args)+total}')
testfn('coder',1,2,3,n1=4,n2=5)

```

Out[14]: 'hello coder, value is 15'

Packing and Unpacking Arguments in Python:

We use two operators * (for tuples) and ** (for dictionaries).

<https://www.geeksforgeeks.org/packing-and-unpacking-arguments-in-python/>

<https://thispointer.com/python-how-to-unpack-list-tuple-or-dictionary-to-function-arguments-using/>

```
In [52]: my_list = [1, 2, 3, 4]
print(*my_list)
```

1 2 3 4

```
In [7]: [*others] = [1,2,3,4]
print(others)
```

```
*others1, = [1,2,3,4]
print(others1)
```

[1, 2, 3, 4]
[1, 2, 3, 4]

```
In [19]: # Consider a situation where we have a function that receives four arguments.
# We want to make call to this function and we have a list of size 4 with us
# that has all arguments for the function. If we simply pass list to the function, the call doesn't work.

# A Python program to demonstrate need of packing and unpacking
def fun(a, b, c, d): # A sample function that takes 4 arguments and prints them.
    print(a, b, c, d)

my_list = [1, 2, 3, 4]
fun(my_list)
```

TypeError

```
Cell In[19], line 10
      7     print(a, b, c, d)
      9 my_list = [1, 2, 3, 4]
----> 10 fun(my_list)
```

Traceback (most recent call last)

TypeError: fun() missing 3 required positional arguments: 'b', 'c', and 'd'

```
In [53]: #We can use * to unpack the list so that all elements of it can be passed as different parameters.
```

A sample function that takes 4 arguments

```
def fun(a, b, c, d):
    print(a, b, c, d)
my_list = [1, 2, 3, 4]
```

Unpacking list into four arguments

```
fun(*my_list)
```

1 2 3 4

```
In [18]: my_list = [[1, 2, 3, 4],[4,5,6],[0,1,2],[7,8,9]]
print(*my_list)
```

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

```
In [19]: def fun(a, b, c, d): # A sample function that takes 4 arguments and prints them.
    print(a, b, c, d)
```

```
my_list = [[1, 2, 3, 4],[4,5,6],[0,1,2],[7,8,9]]
fun(*my_list)
```

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

```
In [20]: my_list = [[1, 2, 3, 4],[4,5,6],[0,1,2],[7,8,9]]
print(*my_list)
```

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

IMP:

```
In [8]: # Packing: using *args, as we don't know how many arguments are going to be passed when the function is called
def fun(*args):
    print(args)
    print(type(args))
fun(1, 2, 3, 4)

print('-----')

# Unpacking:
def fun(a, b, c, d):
    print(a, b, c, d)
my_list = [1, 2, 3, 4]
fun(*my_list) #Unpacking list into four arguments
```

(1, 2, 3, 4)
<class 'tuple'>

1 2 3 4

```
In [41]: # As another example, consider the built-in range() function that expects separate start and stop arguments.
# If they are not available separately,
# write the function call with the *operator to unpack the arguments out of a list or tuple:

range(3, 6) # normal call with separate arguments
print(list(range(3, 6)))

args = [3, 6]
print(list(range(*args))) # call with arguments unpacked from a list
```

[3, 4, 5]
[3, 4, 5]

Packing:

When we don't know how many arguments need to be passed to a function, we can use Packing to pack all arguments in a tuple.

```
In [27]: # This function uses packing to sum unknown number of arguments
def mySum(*args):
    total = 0
    for i in range(0, len(args)):
        total = total + args[i]
    return total

print(mySum(1, 2, 3, 4, 5))
print(mySum(10, 20))
```

15
30

- The above function mySum() does 'packing' to pack all the arguments that this method call receives into one single variable.
- Once we have this 'packed' variable, we can do things with it that we would with a normal tuple.
- args[0] and args[1] would give you the first and second argument, respectively.
- Since our tuples are immutable, you can convert the args tuple to a list so you can also modify, delete and re-arrange items in `i`

```
In [28]: def my(*args):
    for i in range(0, len(args)):
        print(args[i])
    return "all items accessed"
print(my(1, 2, 3, 4, 5))
print(my(10, 20))

1
2
3
4
5
all items accessed
10
20
all items accessed
```

Packing and Unpacking:

```
In [38]: #Below is an example that shows both packing and unpacking.

# A sample python function that takes three arguments and prints them
def fun1(a, b, c):
    print(a, b, c)

# Another sample function.
# This is an example of PACKING. All arguments passed to fun2 are packed into tuple *args.
def fun2(*args):
    # Convert args (tuple) to a list so we can modify it
    args = list(args)
    # Modifying args
    args[0] = 'Geeksforgeeks'
    args[1] = 'awesome'
```

```

print(args) #packed

# UNPACKING args and calling fun1()
fun1(*args)

fun2('Hello', 'beautiful', 'world!')

```

['Geeksforgeeks', 'awesome', 'world!']
Geeksforgeeks awesome world!

```

In [37]: def fun1(a, b, c):
           return (a, b, c)

def fun2(*args):
    args = list(args)
    args[0] = 'Geeksforgeeks'
    args[1] = 'awesome'
    print(args)
    return fun1(*args)

fun2('Hello', 'beautiful', 'world!')

```

['Geeksforgeeks', 'awesome', 'world!']
('Geeksforgeeks', 'awesome', 'world!')

```

Out[37]: In [40]: ls=['Geeksforgeeks', 'awesome', 'world!']
               print(*ls)

Geeksforgeeks awesome world!

```

```

In [43]: *ls=['Geeksforgeeks', 'awesome', 'world!']
          print(ls)

Cell In[43], line 1
  *ls=['Geeksforgeeks', 'awesome', 'world!']
  ^
SyntaxError: starred assignment target must be in a list or tuple

```

```

In [42]: *ls,['Geeksforgeeks', 'awesome', 'world!']
          print(ls)

['Geeksforgeeks', 'awesome', 'world!']

```

```

In [39]: def fun1(a, b, c):
           return (a, b, c)

def fun2(*args):
    args = list(args)
    args[0] = 'Geeksforgeeks'
    args[1] = 'awesome'
    #     print(args)
    return fun1(*args)

fun2('Hello', 'beautiful', 'world!')

```

('Geeksforgeeks', 'awesome', 'world!')

```

In [44]: def fun1(a, b, c):
           print(a, b, c)
def fun2(*args):
    args = list(args)
    args[0] = 'Geeksforgeeks'
    args[1] = 'awesome'
    print(args)
    print(*args)
    fun1(*args)

fun2('Hello', 'beautiful', 'world!')

```

['Geeksforgeeks', 'awesome', 'world!']
Geeksforgeeks awesome world!
Geeksforgeeks awesome world!

```

In [13]: def fun2(*args):
           args = list(args)
           args[0] = 'Geeksforgeeks'
           args[1] = 'awesome'
           print(args)
           print(*args)
           fun1(*args)

fun2('Hello', 'beautiful', 'world!')

```

['Geeksforgeeks', 'awesome', 'world!']
Geeksforgeeks awesome world!

```

In [14]: def fun2(*args):
           print(args[0])
           print(args[1])
           fun1(*args)

fun2('Hello', 'beautiful', 'world!')

```

Hello
beautiful

unpacking of dictionary items using **

In [18]:

```
'''** is used for dictionaries'''
# A sample program to demonstrate unpacking of dictionary items using **
def fun(a, b, c):
    print(a, b, c)

# A call with unpacking of dictionary
d = {'a':2, 'b':4, 'c':10}
fun(**d)
# Here ** unpacked the dictionary used with it, and passed the items in the dictionary as keyword args to the function.
# So writing "fun(1, **d)" was equivalent to writing "fun(1, b=4, c=10)".
```

2 4 10

This code defines a function `fun` that takes three arguments (`a`, `b`, `c`) and prints them. Then, it demonstrates how to call this function using a dictionary `d` with unpacking.

Here's the breakdown:

1. `def fun(a, b, c):` : This line defines a function named `fun` that takes three parameters: `a`, `b`, and `c`.
2. `print(a, b, c)` : This line inside the function prints the values of the parameters `a`, `b`, and `c`.
3. `d = {'a':2, 'b':4, 'c':10}` : This line creates a dictionary `d` with keys `'a'`, `'b'`, and `'c'`, and corresponding values `2`, `4`, and `10`.
4. `fun(**d)` : This line calls the function `fun` with the dictionary `d` unpacked. The `**` operator before `d` unpacks the dictionary into keyword arguments. So, effectively, it's calling `fun(a=2, b=4, c=10)`.

As a result, when you execute `fun(**d)`, it will print `2 4 10`, since these are the values passed into the function `fun` for parameters `a`, `b`, and `c`, respectively.

In [15]:

```
my_tuple=(1,2,3,4)
print(*my_tuple)

d = {'a':2, 'b':4, 'c':10}
print(**d)
```

1 2 3 4

```
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-15-cf0e7810fbca> in <module>
      3
      4 d = {'a':2, 'b':4, 'c':10}
----> 5 print(**d)

TypeError: 'a' is an invalid keyword argument for print()
```

In the provided code:

1. `my_tuple=(1,2,3,4)` : This line creates a tuple named `my_tuple` with four elements: `1`, `2`, `3`, and `4`.
2. `print(*my_tuple)` : The `*` operator before `my_tuple` unpacks the tuple, so it's equivalent to calling `print(1, 2, 3, 4)`. This prints each element of the tuple separated by spaces.

Output: 1 2 3 4

3. `d = {'a':2, 'b':4, 'c':10}` : This line creates a dictionary `d` with keys `'a'`, `'b'`, and `'c'`, and corresponding values `2`, `4`, and `10`.
4. `print(**d)` : This line attempts to unpack the dictionary `d` and pass it as keyword arguments to the `print` function. However, this line will result in a `TypeError` because the `print` function doesn't accept keyword arguments in this way.

You would typically use the `**` operator to unpack a dictionary when calling a function that accepts keyword arguments. For example:

```
print(**{'sep': ' | ', 'end': '\n'})
```

This would unpack the dictionary and pass `'sep'=' | '` and `'end'='\n'` as keyword arguments to the `print` function. However, in the provided code, it's not being used correctly, hence it would raise an error.

So, the output of the provided code will be:

```
1 2 3 4
TypeError: 'a' is an invalid keyword argument for print()
```

** dictionary unpacking is similar to the usage of `**kwargs`

In [6]:

```
def fun(**kwargs):

    # kwargs is a dict
    print(type(kwargs))

    # Printing dictionary items
    for key in kwargs:
```

```

        print(key, kwargs[key])

fun(name="geeks", ID="101", language="Python")

<class 'dict'>
name geeks
ID 101
language Python

```

In [7]:

```

def fun(**kwargs):
    print(type(kwargs))
    for key in kwargs:
        print(key, kwargs[key])
fun(name="geeks", ID="101", language="Python")

<class 'dict'>
name geeks
ID 101
language Python

```

In [2]:

```

def fun(**kwargs):
    print(type(kwargs))
    for key,value in kwargs.items():
        print(key, value)
fun(name="geeks", ID="101", language="Python")

<class 'dict'>
name geeks
ID 101
language Python

```

Exercise:

In [75]:

```

#print highest even number in the input list
def highest_even(x):
    Evens=[]
    for item in x:
        if item%2==0:
            Evens.append(item)
    print(Evens)

    High_Even=[]
    size=len(Evens)
    for i in range(size):
        k=i+1
        for j in range(k,size):
            if Evens[i]>Evens[j]:
                High_Even.append(Evens[i])
    return High_Even

highest_even([2,4,6,10,8,11])

```

Out[75]:

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

In [73]:

```

#print highest even number in the input list
def highest_even(x):
    highest=[]
    for item in x:
        if item%2==0:
            highest.append(item)
    return max(highest)
highest_even([10,2,4,6,8,11])

```

Out[73]:

```
10
```

In [53]:

```

def highest_even(li):
    evens = []
    for item in li:
        if item % 2 == 0:
            evens.append(item)
    return max(evens)

print(highest_even([10,1,2,3,4,8,11]))

```

10

Scope:

Variables can only reach the area in which they are defined, which is called scope.

Think of it as the area of code where variables can be used.

- Python supports global variables (usable in the entire program) and local variables.
- By default, all variables declared in a function are local variables.

In [85]:

```

#scope - what variables do i have access to
x=5      #this variable have global access, means we can acces this variable anywhere in the program.
def fun():

```

```

y=10
print(y) #this variable have Local/function access, means we can acces this variable only inside the function.
fun()

print(x) # x has global access, hence we can print here
print(y) # y has only function access, when we try to access y outside the function, we will get error

10
5
-----
NameError                                 Traceback (most recent call last)
<ipython-input-85-5db05e165d6a> in <module>
      7
      8 print(x) #x has global access, hence we can print here
----> 9 print(y) # y has only function access, when we try to access y outside the function, we will get error

NameError: name 'y' is not defined

```

In [87]:

```

y=10
x=5
def fun():
    print(y)
fun()

print(x) #now we can acess both x,y as they have global access
print(y)

10
5
10

```

Scope rules:

when a function is called, the interpreter searches the required variables in the following order:

1. searches for variable defined in the local/function scope then if variable is not defined in local scope,
2. then searches in parent local scope if variable is not defined in parent local scope,
3. then searches in Global scope if variable is not defined in Global scope,
4. then searches in python built in functions

In [93]:

```

a=1 #global variable
def confusion():
    a=5 #Local variable
    return a

print(a)
confusion()

1
5

```

In [3]:

```

b=1 #global variable
def fun():
    b=10 #parent local variable
    def confusion():
        #Local variable not defined
        return b
    return confusion()

print(b)
fun()

1
10

```

In [4]:

```

b=1 #global variable
def fun():
    #parent Local variable not defined
    def confusion():
        #local variable not defined
        return b
    return confusion()

print(b)
fun()

1
1

```

In [6]:

```

b=1 #global variable
def fun():
    #parent Local variable not defined
    def confusion():
        #local variable not defined
        return mouli
    # we will get error as the variable-mouli is not defined in Local/parent Local/global or is not built-in function
    return confusion()

```

```

print(b)
fun()

1
-----
NameError Traceback (most recent call last)
<ipython-input-6-1fc1df96c84b> in <module>
      8
      9 print(b)
----> 10 fun()

<ipython-input-6-1fc1df96c84b> in fun()
      5     #local variable not defined
      6     return mouli
----> 7     return confusion()
      8
      9 print(b)

<ipython-input-6-1fc1df96c84b> in confusion()
      4     def confusion():
      5         #local variable not defined
----> 6         return mouli
      7         return confusion()
      8

NameError: name 'mouli' is not defined

```

In [7]:

```

b=1 #global variable
def fun():
    #parent local variable not defined
    def confusion():
        #local variable not defined
        return sum
    # we will not get error as the sum is not defined in local/parent local/global scope but sum is a built-in function.
    return confusion()

print(b)
fun()

1
<function sum(iterable, start=0, /)>

```

Out[7]:

In [3]:

```

def fun (a): # parameters of a function are local scope variables.
    print (a)

fun(20)

```

20

In [4]:

```

def fun (a): # parameters of a function are local scope variables.
    print (a)

fun(20)
print(a)

```

20

```

-----
NameError Traceback (most recent call last)
Cell In[4], line 5
      2     print (a)
      4 fun(20)
----> 5 print(a)

NameError: name 'a' is not defined

```

NameError: name 'a' is not defined

global keyword:

In [4]:

```

total=0
def count():
    return total
count()

```

Out[4]:

In [22]:

```

total=0
#here we have delcared 'total' but it is outside the count() function hence it will throw you an error saying -
#you are accessing a variable which is not yet assigned or declared
def count():
    total=total+1
    return total
count()

```

```

-----  

UnboundLocalError                                Traceback (most recent call last)
<ipython-input-22-fd43215b8d68> in <module>
      3     total=total+1
      4     return total
----> 5 count()

<ipython-input-22-fd43215b8d68> in count()
      1 total=0
      2 def count():
----> 3     total=total+1
      4     return total
      5 count()

UnboundLocalError: local variable 'total' referenced before assignment

```

```
In [21]: def count():
    total=0
    total=total+1
    return total
print(count())
print(count())
print(count())
```

```
1
1
1
```

```
In [23]: total=0
def count():
    global total
    total=total+1
    return total
count()
```

```
Out[23]: 1
```

```
In [20]: total=0
def count():
    global total
    total=total+1
    return total
print(count())
print(count())
print(count())
```

```
1
2
3
```

```
In [2]: def count(total):
    total=total+1
    return total
print(count(count(count(0))))
```

```
3
```

nonlocal keyword:

nonlocal keyword doesn't refer to the global scope, it refers to the variable outside the local/function scope i.e., parent local

1. https://www.w3schools.com/python/ref_keyword_nonlocal.asp
2. <https://www.geeksforgeeks.org/use-of-nonlocal-vs-use-of-global-keyword-in-python/>

```
In [22]: def outer():
    x='hi'
    def inner():
        nonlocal x #x='hi'
        x='hello' #after execution of this statement, x gets modified to x='hello' from x='hi'(nonLocal x got modified)
        print('inner:',x) #x='hello'
    inner()
    print("outer:",x)#x='hello', as it got modified after execution of line5

outer()

inner: hello
outer: hello
```

```
In [23]: def outer():

    x='hi'
    def inner():
        #nonLocal x
        x='hello'
        print('inner:',x)# x='hello'
    inner()

    print("outer:",x)#x='hi'

outer()
```

```
inner: hello
outer: hi
```

```
In [1]: # Scope - what variables do I have access to?
def outer():
    x = "local"
    def inner():
        nonlocal x
        x = "nonlocal"
        print("inner:", x)
    inner()
    print("outer:", x)
outer()

#variable search order:
#1 - start with local
#2 - Parent Local
#3 - global
#4 - built in python functions
```

```
inner: nonlocal
outer: nonlocal
```

```
In [5]: # Scope - what variables do I have access to?
def outer():
    x = 10
    def inner():
        nonlocal x
        x = 20
        print("inner:", x)
    inner()
    print("outer:", x)
outer()

#variable search order:
#1 - start with local
#2 - Parent Local
#3 - global
#4 - built in python functions
```

```
inner: 20
outer: 20
```

```
In [7]: x = 10 # global variable
def outer():
    y = 20 # enclosing scope variable
    def inner():
        nonlocal y # nonlocal keyword
        y = 30 # this will change the value of y in outer() scope, not local
        z = 40 # local variable to inner() function
        print(f"inner: {z}")
    def innermost():
        global x # nonlocal keyword
        nonlocal z # global keyword
        x = 50 # this will change the value of x in global scope, not enclosing or local
        z = 60 # this will change the value of z in outer() scope, not local
        print(f"innermost: {x}, {y}, {z}")
    innermost()
    print(f"inner after innermost: {x}, {y}, {z}")
    inner()
    print(f"outer after inner: {x}, {y}")
outer()
print(f"global after outer: {x}")

inner: 40
innermost: 50, 30, 60
inner after innermost: 50, 30, 60
outer after inner: 50, 30
global after outer: 50
```

Walrus Operator:

It assigns value to variables as part of a larger expression

```
In [3]: a='walrusoperator'
if len(a)>5:
    print(f"The length of string is {len(a)} which is > 5")
```

```
The length of string is 14 which is > 5
```

```
In [5]: a='walrusoperator'
if (n:= len(a))>5: #it assigns value to variables as part of a larger expression.
    print(f"The length of string is {n} which is > 5")
```

```
File "<ipython-input-5-056bb90f17f2>", line 2
  if (n:= len(a))>5: #it assigns value to variables as part of a larger expression.
  ^
SyntaxError: invalid syntax
```

```
In [4]: a='walrusoperator'
if (n:= len(a))>5: #it assigns value to variables as part of a larger expression.
    print(f"The length of string is {n} which is > 5")
```

The length of string is 14 which is > 5

```
In [8]: a='walrusoperator'
while ((n := len(a)) > 1):
    print(n)
    a = a[0:-1] # removes one element everytime
    print(a)
print(a)

14
walrusoperator
13
walrusoperat
12
walrusopera
11
walrusoper
10
walrusope
9
walrusop
8
walruso
7
walrus
6
walru
5
walr
4
wal
3
wa
2
w
w
```

`while ((n := len(a)) > 1):` This line starts a while loop that continues as long as the length of a (minus 1) is greater than 1. The `:=` operator is called the "walrus operator" because it assigns a value to a variable as part of an expression. In this case, it assigns the length of a to the variable n, then checks if n is greater than 1.

```
In [3]: l='krish'
l[0:-1]
```

```
Out[3]: 'kris'
```

Advanced Python: Object Oriented Programming

OOPS:

Everything here is an object because in Python everything is built by this class keyword and we're able to use different methods on our objects like this to perform some actions on them. Objects have methods and attributes that you can access with the DOT method.

An object is simply a collection of data (variables) and methods (functions) that act on those data.

- https://www.w3schools.com/python/python_classes.asp
- <https://www.geeksforgeeks.org/python-classes-and-objects/>

OOP is what we call a paradigm that is, it's a way for us to think about our code and structure our code in a way that is easier to maintain extend and write.

```
In [2]: print(type(None))
print(type(True))
print(type(5))
print(type(5.5))
print(type('hi'))
print(type([]))
print(type(()))
print(type({}))
```

```
<class 'NoneType'>
<class 'bool'>
<class 'int'>
<class 'float'>
<class 'str'>
<class 'list'>
<class 'tuple'>
<class 'dict'>
```

Class name should be always singular

```
In [1]: #In python we can create our own datatype i.e., own class
class BigObject:
    pass
```

```
print(type(BigObject))
<class 'type'>
```

We can now create let's say object equals to big object and then run this class. so that if I type object here Object 1 and I click Run, I get class big object.

```
In [3]: class BigObject: #class
    #code
    pass
obj1=BigObject() #object and () means instantiating
#we are creating an instance/object(obj1) by instantiating the class.
obj2=BigObject() #object
obj3=BigObject() #object
print(type(obj1))

<class '__main__.BigObject'>
```

Now a class is this:

- It's the blueprint, the blueprint of what we want to create.
- What are the basic attributes that is properties that our class has.
- What are some basic methods or actions that our class can take.
- And then from this blueprint I'm able to create different objects over and over using this as the building block.

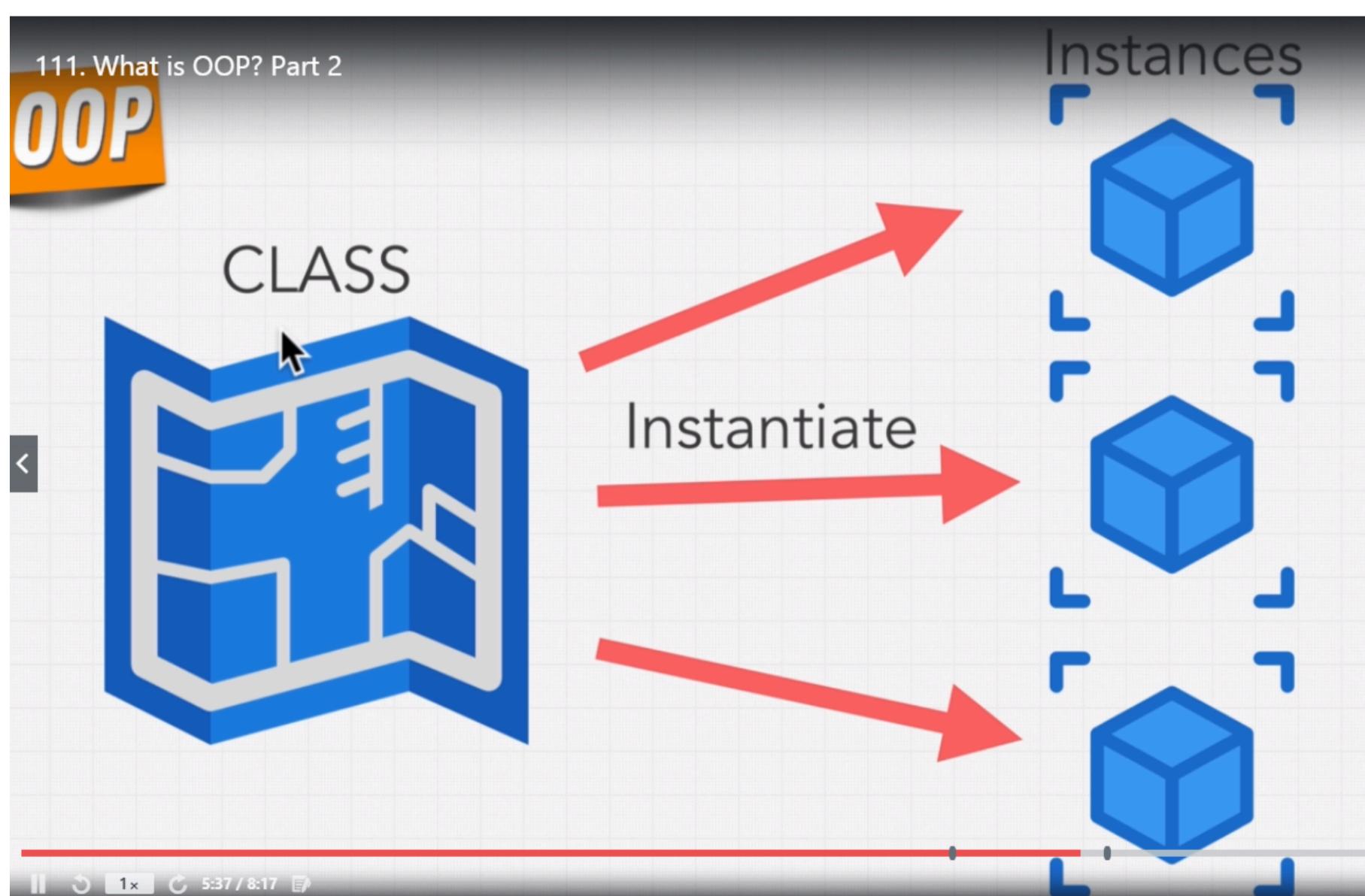
So this blueprint is what we call a class which we define with the class keyword and

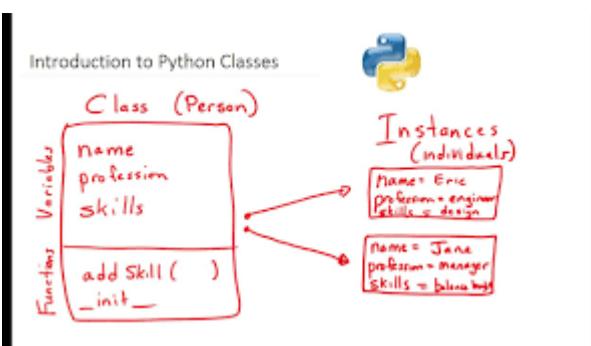
- this class can be instantiated-That is the action of creating different instances.
- And what are these instances-These are all objects.

We've created here a class or a blueprint (class BigObject:) and then here this double bracket in line 4 is instantiating the class and saying Hey class use whatever you have code in class and instantiated and create a new object.

So we're creating a new object (obj1=BigObject()) by instantiating class (i.e., class is BigObject) now I have three different objects that I can use based on this blueprint.

The class is going to be stored in memory so Python interpreter is going to say hey big object it's going to be the blueprint for this. So I'm going to store all that code in memory but every time I create an object I don't have to rewrite the code or do anything like that. I can simply say hey go in memory to where bigobject is and just run that code so that again we're keeping our code dry.





```
In [9]: class PlayerCharacter:

    #code block start
    def __init__(self, name): # self refers to the PlayerCharacter class.
        self.name = name
    #code block end

    def run(self): #run() is a class method.
        print("run")

player1 = PlayerCharacter()
#when I tried to instantiate the player character it runs __init__ and then it tries to do self.name equals to name.
print(player1)
```

```
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-9-0cada2b4a640> in <module>
      9         print("run")
     10
----> 11 player1 = PlayerCharacter()
     12 #when I tried to instantiate the player character it runs __init__ and then it tries to do self.name equals to name.
     13 print(player1)

TypeError: __init__() missing 1 required positional argument: 'name'
```

`__init__` method is a special method.

You see the two underlines here. This is called a `Dunder method` or a `magic method`.

When we're building a class you usually see `__init__` defined at the top and this is what's often called a `constructor method` or `init method` and this is automatically called anytime we instantiate an object.

remember instantiate means, we're calling the class to create an object. So when I do this it's going to automatically run whatever is in this code block

In Python, the `__init__` method in a class is called automatically when an object (instance) of that class is created. It is used to initialize the attributes of the class.

You don't necessarily have to write a `__init__` method in every class you create. It is only required when you need to perform some initialization when an object is created. If there's nothing to initialize, you don't need an `__init__` method.

Here's a simple example:

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

    def introduce(self):
        print(f"Hello, my name is {self.name} and I'm {self.age} years old.")

p = Person("John", 30) # Creates an object of the class Person
p.introduce() # Outputs: Hello, my name is John and I'm 30 years old.
```

In this example, the `__init__` method is used to initialize the attributes `name` and `age` of the `Person` class when an object is created. If we didn't have the `__init__` method, we wouldn't be able to set the `name` and `age` attributes when creating the object `p`.

In Python, the `__init__` method is not required in all scenarios. Here's an example where it's not needed:

```
class Car:
    def drive(self):
        print("The car is driving.")

c = Car() # Creates an object of the class Car
c.drive() # Outputs: The car is driving.
```

In this example, we don't need an `__init__` method because we're not initializing any attributes when creating the `Car` object. The `drive` method doesn't depend on any instance attributes, so it can be called without any problems even though there's no `__init__` method.

```
In [10]: class PlayerCharacter:

    def __init__(self, name):
```

```

    self.name=name

    def run(self):
        print("run")

player1=PlayerCharacter("candy")
print(player1) #shows that the object player1 is at which memory location.

<__main__.PlayerCharacter object at 0x000000338D225408>

```

In [12]:

```

class PlayerCharacter:

    def __init__(self, name):
        self.name=name

    def run(self):
        print("run")

player1=PlayerCharacter("candy")
print(player1.name)# we can access name/age outside the class using objectname.attribute i.e., player1.name

candy

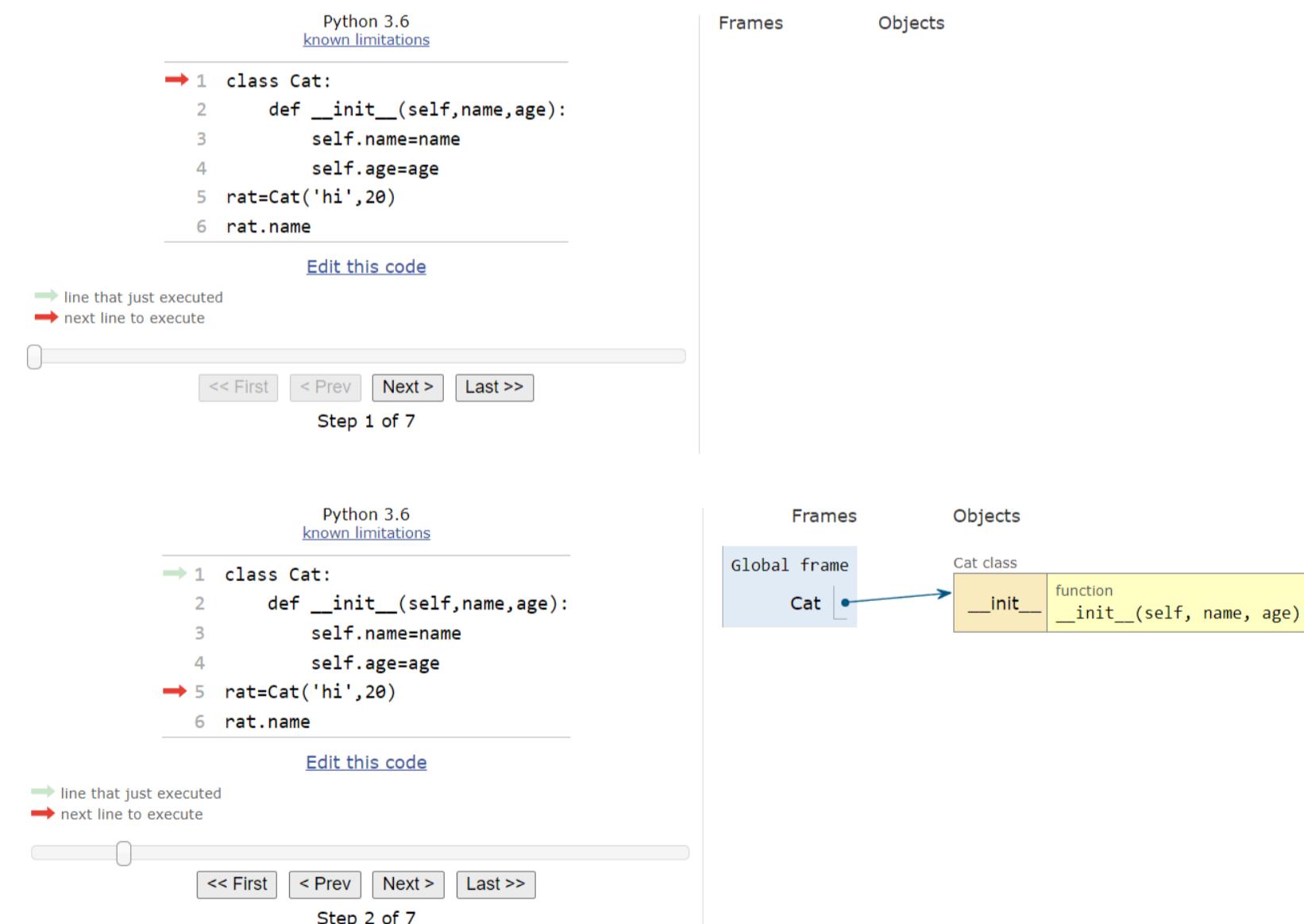
```

`self` refers to the PlayerCharacter class.

It's saying hey `self` in `(def __init__(self, name))` is going to refer to this PlayerCharacter that we're going to create Player 1 and I want `self.name` to equal whatever the parameter name is so that if I do `print(player.name)` and I click Run I get Candy and why is it Candy. Well because I gave the argument in `playercharacter` in line 10 and when I instantiate it, I passed the name parameter.

The default parameter is `self` when I do `__init__` and in classes you see here that always the first parameter is default when we're defining a method and it is `self` and now I give `name` to `self.name`.

`self` is default parameter in classes.



Python 3.6
[known limitations](#)

```

1 class Cat:
2     def __init__(self, name, age):
3         self.name=name
4         self.age=age
5 rat=Cat('hi',20)
6 rat.name

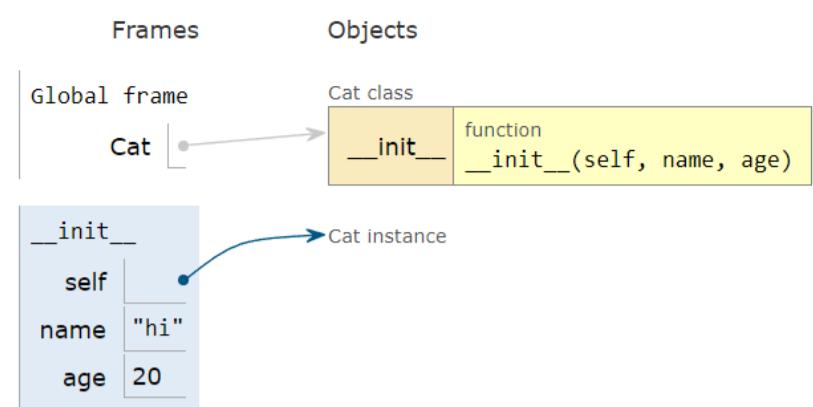
```

[Edit this code](#)

Line status: green arrow line that just executed red arrow next line to execute

[<< First] [< Prev] [Next >] [Last >>]

Step 3 of 7



Python 3.6
[known limitations](#)

```

1 class Cat:
2     def __init__(self, name, age):
3         self.name=name
4         self.age=age
5 rat=Cat('hi',20)
6 rat.name

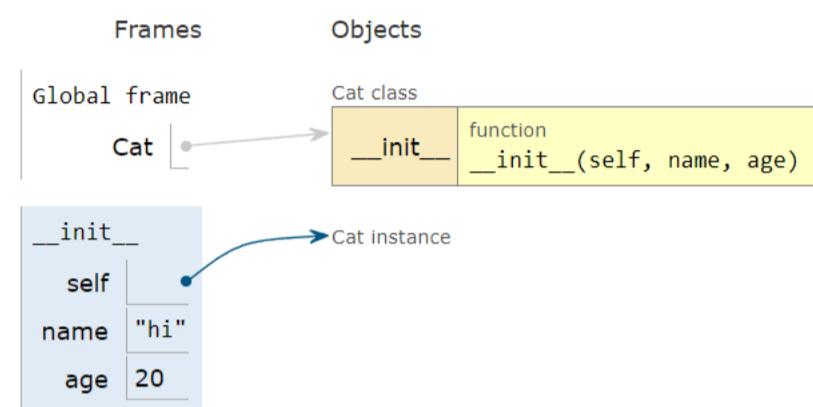
```

[Edit this code](#)

Line status: green arrow line that just executed red arrow next line to execute

[<< First] [< Prev] [Next >] [Last >>]

Step 4 of 7



Python 3.6
[known limitations](#)

```

1 class Cat:
2     def __init__(self, name, age):
3         self.name=name
4         self.age=age
5 rat=Cat('hi',20)
6 rat.name

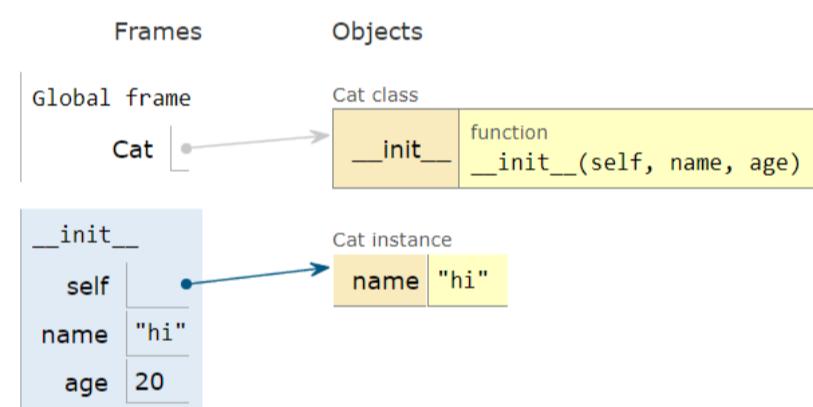
```

[Edit this code](#)

Line status: green arrow line that just executed red arrow next line to execute

[<< First] [< Prev] [Next >] [Last >>]

Step 5 of 7



Python 3.6
[known limitations](#)

```

1 class Cat:
2     def __init__(self, name, age):
3         self.name=name
4         self.age=age
5 rat=Cat('hi',20)
6 rat.name

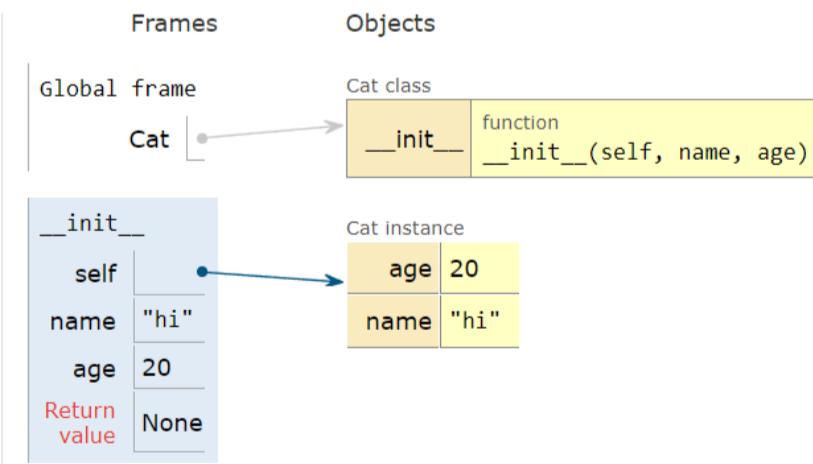
```

[Edit this code](#)

Line status: green arrow line that just executed red arrow next line to execute

[<< First] [< Prev] [Next >] [Last >>]

Step 6 of 7



Python 3.6
[known limitations](#)

```

1 class Cat:
2     def __init__(self, name, age):
3         self.name=name
4         self.age=age
5 rat=Cat('hi',20)
6 rat.name

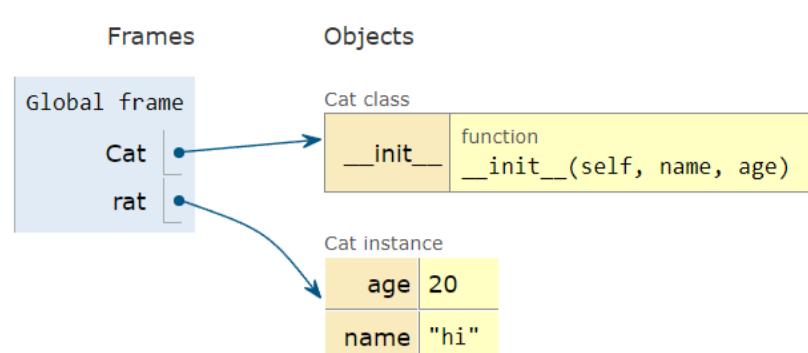
```

[Edit this code](#)

Line status: green arrow line that just executed red arrow next line to execute

<< First < Prev Next > >> Last

Step 7 of 7



Python 3.6
[known limitations](#)

```

1 class Cat:
2     def __init__(self, name, age):
3         self.name=name
4         self.age=age
5 rat=Cat('hi',20)
6 rat.name

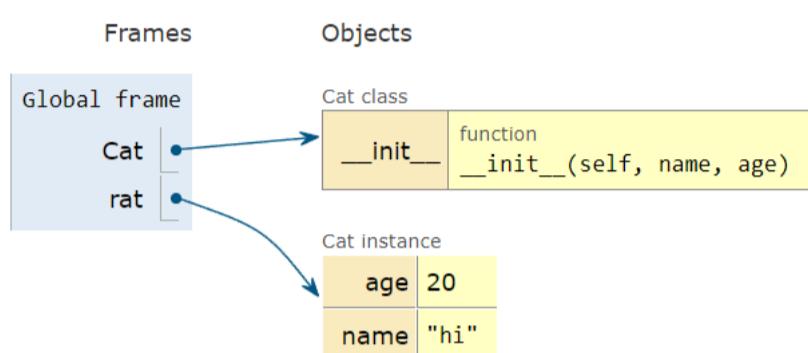
```

[Edit this code](#)

Line status: green arrow line that just executed red arrow next line to execute

<< First < Prev Next > >> Last

Done running (7 steps)



In [3]:

```

class PlayerCharacter:
    def __init__(self, name):
        name=name
    def run(self):
        print("run")
player1=PlayerCharacter("candy")
print(player1.name)

# What if I remove self here and I click Run you see that it says object has no attribute name
# because in order for me to say hey I want player to have a name. I need to say self dot because self refers to PlayerCharacter.

```

AttributeError Traceback (most recent call last)
Cell In[3], line 7
5 print("run")
6 player1=PlayerCharacter("candy")
----> 7 print(player1.name)

AttributeError: 'PlayerCharacter' object has no attribute 'name'

self allows us to have a reference to something that hasn't been created yet.

In this case Player 1 and let them know that hey when player1 is created and when we instantiate, then I want you to make sure that player 1 has this name attribute.

In [22]:

```

class PlayerCharacter:
    def __init__(self, name):
        self.name=name
    def run(self):
        print("run")

player1=PlayerCharacter("candy")
player2=PlayerCharacter("tom")
print(player1.name)
print(player2.name)

```

candy
tom

In [4]:

```

class PlayerCharacter:
    def __init__(self, name):
        self.name=name
player1=PlayerCharacter("candy")
player2=PlayerCharacter("tom")
print(player1.name)
print(player2.name)

```

candy
tom

```
In [25]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name #attributes
        self.age=age

    def run(self):
        print("run")
```

```
player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
print(player1.name)
print(player1.age)
print(player2.name)
```

#Here name and age are the attributes/properties that the objects/instances(player1,player2) have.

```
candy
22
tom
```

```
In [32]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name
        self.age=age
```

```
def run(self):
    print("run")

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
# print(player1.name)
print(player1.run())
print(player2.name)
```

```
<bound method PlayerCharacter.run of <__main__.PlayerCharacter object at 0x000000338D238648>>
tom
```

```
In [8]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self):#run() is a method
        print("run")
```

```
player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)

print(player1.run())
print(player2.name)
```

```
run
None
tom
```

```
In [9]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self):#run() is a method
        print("run")
```

```
player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)

print(player1.run())
```

```
run
None
```

```
In [6]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self):#run() is a method
        return "run"
```

```
player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
print(player1.run())
print(player2.name)
```

```
run
tom
```

```
In [38]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self):
```

```

        print("run")
        return "done"

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
print(player2.name)
print(player2.attack)
print(player1.attack) # attribute error as player1 object doesn't have attack attribute.

```

tom
52

```

-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-38-da9ba1c7a1a1> in <module>
      14     print(player2.name)
      15     print(player2.attack)
----> 16     print(player1.attack)

AttributeError: 'PlayerCharacter' object has no attribute 'attack'

```

Attributes and Methods:

```
In [42]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self):
        print("run")
        return "done"

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
help(player2)
```

Help on PlayerCharacter in module __main__ object:

```
class PlayerCharacter(builtins.object)
| PlayerCharacter(name, age)
|
| Methods defined here:
|
|     __init__(self, name, age)
|         Initialize self. See help(type(self)) for accurate signature.
|
|     run(self)
|
| -----
| Data descriptors defined here:
|
|     __dict__
|         dictionary for instance variables (if defined)
|
|     __weakref__
|         list of weak references to the object (if defined)
```

```
In [43]: class PlayerCharacter:
    playermembership=True # Class Object Attribute
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self):
        print("run")
        return "done"

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
print(player2.playermembership) # we can access class attributes
```

True

```
In [10]: class PlayerCharacter:
    playermembership=True # Class Object Attribute
    def __init__(self, name, age):
        if (playermembership): # NameError: name 'playermembership' is not defined
            self.name=name
            self.age=age

    def run(self):
        print("run")
        return "done"

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
```

```

print(player1.playermembership)
print(player1.name)

-----
NameError                                                 Traceback (most recent call last)
<ipython-input-10-72b4191ce2f1> in <module>
      10         return "done"
      11
---> 12 player1=PlayerCharacter("candy",22)
      13 player2=PlayerCharacter("tom",41)
      14 player2.attack=52

<ipython-input-10-72b4191ce2f1> in __init__(self, name, age)
      2     playermembership=True #Class Object Attribute
      3     def __init__(self,name,age):
----> 4         if (playermembership):
      5             self.name=name
      6             self.age=age

NameError: name 'playermembership' is not defined

```

In [5]:

```

class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            # we can access playermembership attribute by self.playermembership or PlayerCharacter.playermembership
            self.name=name
            self.age=age

    def run(self):
        print("run")
        return "done"

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52

print(player2.playermembership)
print(player1.name)
print(player2.name)

```

True
candy
tom

In [6]:

```

class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (PlayerCharacter.playermembership):
            #we can access playermembership attribute by self.playermembership or
            #PlayerCharacter.playermembership as playermembership is Class object attribute.
            self.name=name
            self.age=age

    def run(self):
        print("run")
        return "done"

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52

print(player2.playermembership)
print(player1.name) # we can access name/age outside the class using objectname.attribute i.e., player1.name
print(player2.name)

```

True
candy
tom

In [8]:

```

class PlayerCharacter:
    playermembership=False #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            self.name=name # these attributes are defined under the condition of playembership to be true
            self.age=age

    def run(self):
        print("run")
        return "done"

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52

print(player2.playermembership)
print(player1.name) # as the playermembership is false, we cannot access the name attribute.
print(player2.name)

```

False

```

-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-8-352fb1785e95> in <module>
    15
    16     print(player2.playermembership)
--> 17     print(player1.name)
    18     print(player2.name)

AttributeError: 'PlayerCharacter' object has no attribute 'name'

```

```

In [10]: class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            self.name=name
            self.age=age

    def run(self):
        print(f"My name is {name}")

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
print(player1.run())
print(player2.run())

```

```

-----
NameError                                     Traceback (most recent call last)
<ipython-input-10-feabb2457d3b> in <module>
    13     player2=PlayerCharacter("tom",41)
    14     player2.attack=52
--> 15     print(player1.run())
    16     print(player2.run())

<ipython-input-10-feabb2457d3b> in run(self)
    7
    8     def run(self):
--> 9         print(f"My name is {name}")
    10
    11

NameError: name 'name' is not defined

```

IMP:

1. we can access name/age inside the class using self.name/self.age
2. we can access name/age outside the class using objectname.attribute i.e., player1.name

```

In [7]: class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            self.name=name
            self.age=age

    def run(self):
        print(f"My name is {self.name}") # we can access name/age inside the class using self.name/self.age

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
print(player1.name) # we can access name/age outside the class using objectname.attribute i.e., player1.name
print(player1.run())
print(player2.run())

```

candy
My name is candy
None
My name is tom
None

```

In [26]: class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            self.name=name
            self.age=age

    def run(self):
        return(f"My name is {self.name}")
        #print(f"My age is {self.age}")

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
print(player1.run())
print(player2.run())

```

```
My name is candy
My name is tom
```

```
In [19]: class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            self.name=name
            self.age=age

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
print(f'My name is {player1.name}')
print(f'My name is {player2.name}')
```

```
My name is candy
My name is tom
```

```
In [55]: class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            self.name=name
            self.age=age

    def fun(self):
        return(f"My name is {self.name}")
        #print(f"My age is {self.age}")

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
print(player1.fun())
print(player2.fun())
```

```
My name is candy
My name is tom
```

```
In [11]: class Test:
    att=True
    def __init__(self,name):
        if self.att:
            self.name=name
p1=Test('user')
p1.name
```

```
Out[11]: 'user'
```

```
In [10]: class Test:
    att=False
    def __init__(self,name):
        if self.att:
            self.name=name
p1=Test('user')
p1.name
```

```
-----
AttributeError                                     Traceback (most recent call last)
Cell In[10], line 7
      5             self.name=name
      6 p1=Test('user')
----> 7 p1.name

AttributeError: 'Test' object has no attribute 'name'
```

My Findings: (6 code blocks below)

```
In [41]: class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            self.name=name
            self.age=age
    def run(coder):
        return(f"My name is {name}")
    # NameError: name 'name' is not defined -- we can access name/age inside the class using self.name/self.age
player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
print(player1.run())
print(player2.run())
```

```
NameError Traceback (most recent call last)
<ipython-input-41-c066d9640cc4> in <module>
    10 player2=PlayerCharacter("tom",41)
    11 player2.attack=52
---> 12 print(player1.run())
    13 print(player2.run())

<ipython-input-41-c066d9640cc4> in run(coder)
     6         self.age=age
     7     def run(coder):
---> 8         return(f"My name is {name}")
     9 player1=PlayerCharacter("candy",22)
    10 player2=PlayerCharacter("tom",41)

NameError: name 'name' is not defined
```

```
In [42]: class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            self.name=name
            self.age=age
    def run(coder):
        return(f"My name is {self.name}") # NameError: name 'self' is not defined
    # self is the default parameter for all class methods
player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
print(player1.run())
print(player2.run())
```

```
NameError Traceback (most recent call last)
<ipython-input-42-a26103701d4b> in <module>
    10 player2=PlayerCharacter("tom",41)
    11 player2.attack=52
---> 12 print(player1.run())
    13 print(player2.run())

<ipython-input-42-a26103701d4b> in run(coder)
     6         self.age=age
     7     def run(coder):
---> 8         return(f"My name is {self.name}")
     9 player1=PlayerCharacter("candy",22)
    10 player2=PlayerCharacter("tom",41)

NameError: name 'self' is not defined
```

```
In [43]: class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            self.name=name
            self.age=age
    def run(self):
        return(f"My name is {self.name}")
player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
print(player1.run())
print(player2.run())
```

My name is candy
My name is tom

```
In [49]: class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            self.name=name
            self.age=age

    def run(self,naam): #self is a default parameter of class and other parameter which is naam.
        return(f"My name is {self.name}")

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
print(player1.run())
print(player2.run())
```

```
TypeError Traceback (most recent call last)
<ipython-input-49-5b13cae08f9c> in <module>
    12 player2=PlayerCharacter("tom",41)
    13 player2.attack=52
---> 14 print(player1.run())
    15 print(player2.run())

TypeError: run() missing 1 required positional argument: 'naam'
```

In [50]:

```

class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            self.name=name
            self.age=age

    def run(self,naam):    #self is a default parameter of class, so here run counts only 1parameter which is naam.
        return(f"My name is {self.naam}")

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
print(player1.run("joe"))
print(player2.run("marie"))

```

```

-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-50-fb370704d917> in <module>
      12 player2=PlayerCharacter("tom",41)
      13 player2.attack=52
---> 14 print(player1.run("joe"))
      15 print(player2.run("marie"))

<ipython-input-50-fb370704d917> in run(self, naam)
      7
      8     def run(self,naam):    #self is a default parameter of class, so here run counts only 1parameter which is name.
----> 9         return(f"My name is {self.naam}")
     10
     11 player1=PlayerCharacter("candy",22)

AttributeError: 'PlayerCharacter' object has no attribute 'naam'

```

In [58]:

```

class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,naam,age):
        if (self.playermembership):
            self.name=name
            self.naam=naam
            self.age=age

    def run(self,naam):    #self is a default parameter of class, so here run counts only 1parameter which is name.
        return(f"My name is {self.naam}")

player1=PlayerCharacter("candy",'can',22)
player2=PlayerCharacter("tom",'tm',41)
player2.attack=52
print(player1.run("joe"))
#here we will get 'can' because player1 is an object of PlayerCharacter and we gave ("candy",'can',22) as arguments there.
print(player2.run("marie"))
#here we will get 'tm' because player1 is an object of PlayerCharacter and we gave("tom",'tm',41) as arguments there.

```

My name is can
My name is tm

In [12]:

```

class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            self.name=name
            self.age=age

    def run(self,naam):    #self is a default parameter of class, so here run counts only 1parameter which is naam.
        return(f"My name is {naam}")
    #here naam is not a class attribute so we can directly call it

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52
print(player1.run("joe"))
print(player2.run("marie"))

```

My name is joe
My name is marie

- we can access name/age inside the class using self.name/self.age
- we can access name/age outside the class using objectname.attribute i.e., player1.name

In the above code naam is not a attribute defined in the class so we can directly call it

In [59]:

```

class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,naam,age):
        if (self.playermembership):
            self.name=name
            self.naam=naam
            self.age=age

    def run(self):
        return(f"My name is {self.age}")

```

```

def hello(self,naam):
    return(f"My name is {self.naam}")

player1=PlayerCharacter("candy",'can',22)
player2=PlayerCharacter("tom",'tm',41)
player2.attack=52

print(player1.run("joe"))
print(player2.hello("marie"))

```

```

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-59-ec4c08f832a5> in <module>
      17 player2.attack=52
      18
---> 19 print(player1.run("joe"))
      20 print(player2.hello("marie"))

TypeError: run() takes 1 positional argument but 2 were given

```

```

In [62]: class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,naam,age):
        if (self.playermembership):
            self.name=name
            self.naam=naam
            self.age=age

    def run(self):
        return(f"My age is {self.age}")

    def hello(self,naam):
        return(f"My name is {self.naam}")

player1=PlayerCharacter("candy",'can',22)
player2=PlayerCharacter("tom",'tm',41)
player2.attack=52

print(player1.run())
print(player2.hello("marie"))

```

My age is 22
My name is tm

IMP:

Instance Attribute and Class Attribute

```

In [13]: class PlayerCharacter:
    playermembership=True #Class Object Attribute
    def __init__(self,name,age):
        if (self.playermembership):
            PlayerCharacter.name=name
            PlayerCharacter.age=age
    def run(self):
        print("run")
        return "done"

player1=PlayerCharacter("candy",22)
player2=PlayerCharacter("tom",41)
player2.attack=52

print(player2.playermembership)
print(player1.name)
print(player1.age)
print(player2.name)
print(player1.age)

```

True
tom
41
tom
41

Given `PlayerCharacter.name=name` `PlayerCharacter.age=age`

instead of `self.name=name` `self.age=age`

By doing it, You are changing name,age from an instance attribute to a class attribute,

- So when you create player1 it gets set to candy and when you create player2 it gets set to tom.
- Because it's being created as a class attribute, it's the same for all objects of the class.

`__init__` method/constructor method:

```

In [37]: class PlayerCharacter: #class
    playermembership=True #Class Object Attribute

```

```

def __init__(self, name, age): #constructor method .....this gets called everytime we instanciate an object
    if (self.playermembership):
        self.name=name #attribute
        self.age=age
    def run(self): # run is a class method
        return(f"My name is {self.name}")
player1=PlayerCharacter("candy",22) #object
player2=PlayerCharacter("tom",41)
player2.attack=52
print(player1.run())
print(player2.run())

```

My name is candy
My name is tom

```

In [66]: class PlayerCharacter: #class
    playermembership=True #Class Object Attribute
    def __init__(self, name, age): #constructor method .....this gets called everytime we instanciate an object
        if (age>18):
            self.name=name #attribute
            self.age=age
        def run(self): # run is a class method
            return(f"My name is {self.name}")
player1=PlayerCharacter("candy",22) #object
player2=PlayerCharacter("tom",17)
player2.attack=52
print(player1.name)
print(player2.name) # throws an error as the age is <18

```

candy

```

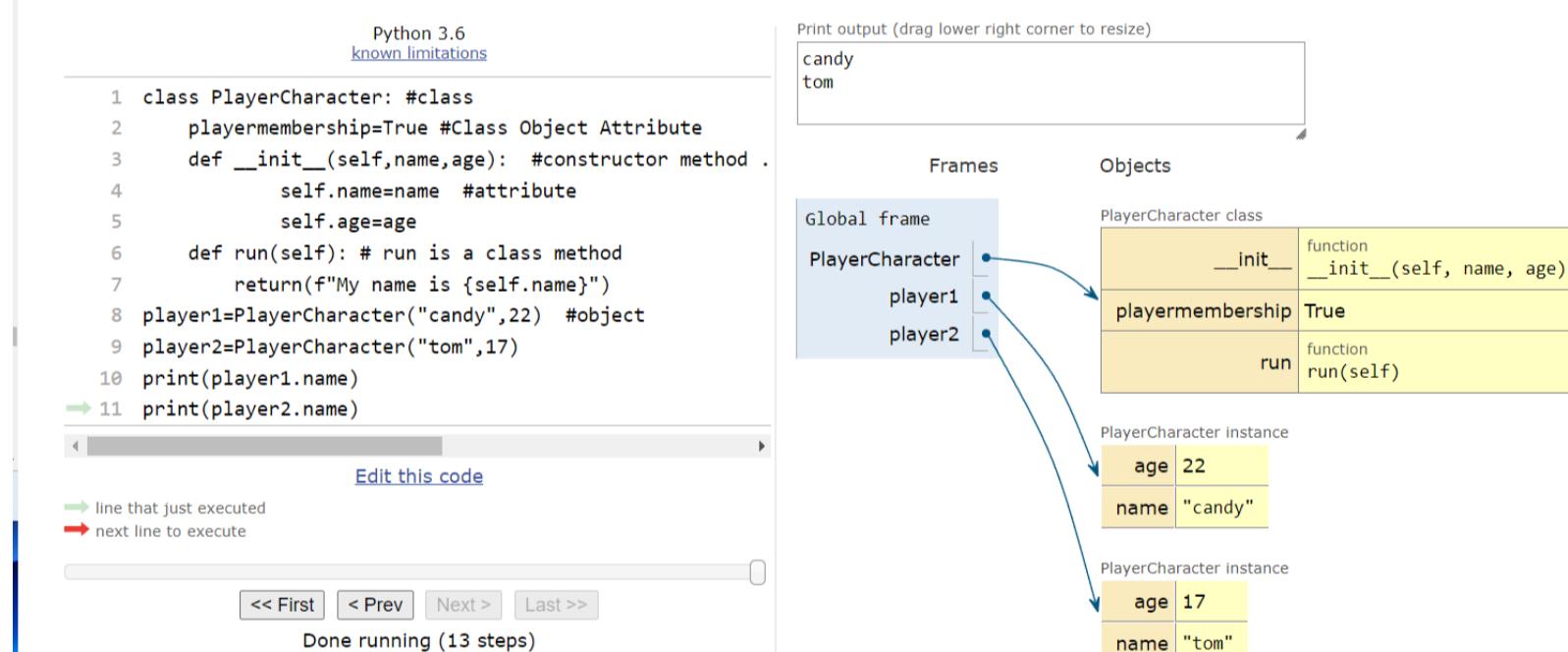
-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-66-7c5fac2ce30f> in <module>
      11 player2.attack=52
      12 print(player1.name)
---> 13 print(player2.name) # throws an error as the age is <18

AttributeError: 'PlayerCharacter' object has no attribute 'name'

```

[← → ⌂ pythontutor.com/render.html#mode=display](http://pythontutor.com/render.html#mode=display)

Python Tutor: Visualize code in Python, JavaScript, C, C++, and Java



The flow of execution - Class, Objects:

```

In [67]: class PlayerCharacter: #class
    playermembership=True #Class Object Attribute
    def __init__(self, name="anonymous", age): #constructor method .....this gets called everytime we instanciate an object
        if (age>18):
            self.name=name #attribute
            self.age=age
    def run(self): # run is a class method
        return(f"My name is {self.name}")
player1=PlayerCharacter("candy",22) #object
player2=PlayerCharacter("tom",17)
player2.attack=52
print(player1.name)
print(player2.name)

```

```

File "<ipython-input-67-afdc318843ec>", line 3
  def __init__(self, name="anonymous", age): #constructor method .....this gets called everytime we instanciate an object
  ^
SyntaxError: non-default argument follows default argument

```

Got error as we defined parameters in wrong way. remember the order of defining the parameters:

(parms,args,default parms,*kwargs)

```
In [6]: class PlayerCharacter: #class
    playermembership=True #Class Object Attribute
    def __init__(self,age,name="anonymous"): #constructor method .....this gets called everytime we instanciate an object
        if (age>=18):
            self.name=name #attribute
            self.age=age
    def run(self): # run is a class method
        return(f"My name is {self.name}")
player1=PlayerCharacter(22,"candy") #object
player2=PlayerCharacter(18,"tom")
player2.attack=52
print(player1.name)
print(player2.name)
```

candy
tom

```
In [14]: class PlayerCharacter: #class
    playermembership=True #Class Object Attribute
    def __init__(self,name="anonymous",age=95):#constructor method .....this gets called everytime we instanciate an object
        self.name=name #attribute
        self.age=age
    def run(self): # run is a class method
        return(f"My name is {self.name}")

player3=PlayerCharacter(age=29) #object
print(player3.name)
print(player3.age)

class PlayerCharacter: #class
    playermembership=True #Class Object Attribute
    def __init__(self,age=66,name="anonymous"):#constructor method .....this gets called everytime we instanciate an object
        self.name=name #attribute
        self.age=age
    def run(self): # run is a class method
        return(f"My name is {self.name}")

player2=PlayerCharacter(name="eion") #object
print(player2.name)
print(player2.age)
```

anonymous
29
eion
66

without age restriction:

```
In [15]: class PlayerCharacter: #class
    playermembership=True #Class Object Attribute
    def __init__(self,name="anonymous",age=95):
        self.name=name #attribute
        self.age=age
    def run(self): # run is a class method
        return(f"My name is {self.name}")
player1=PlayerCharacter("lranon",25) #object
player2=PlayerCharacter("eion") #object
player3=PlayerCharacter(age=29) #object
player4=PlayerCharacter()

print(player1.name)
print(player1.age)
print(player2.name)
print(player2.age)
print(player3.name)
print(player3.age)
print(player4.name)
print(player4.age)
```

lranon
25
eion
95
anonymous
29
anonymous
95

```
In [1]: class PlayerCharacter: #class
    playermembership=True #Class Object Attribute
    def __init__(self,age=66,name="anonymous"):
        self.name=name #attribute
        self.age=age
    def run(self): # run is a class method
        return(f"My name is {self.name}")
player1=PlayerCharacter(25,"lranon") #object
player2=PlayerCharacter(name="eion") #object
player3=PlayerCharacter(29) #object
player4=PlayerCharacter()

print(player1.name)
print(player1.age)
print(player2.name)
```

```
print(player2.age)
print(player3.name)
print(player3.age)
print(player4.name)
print(player4.age)
```

```
lranon
25
eion
66
anonymous
29
anonymous
66
```

with age restriction:

```
In [4]: class PlayerCharacter: #class
    playermembership=True #Class Object Attribute
    def __init__(self, name="anonymous", age=95):
        if (age>=18):
            self.name=name #attribute
            self.age=age
    def run(self): # run is a class method
        return(f"My name is {self.name}")
player1=PlayerCharacter("lranon",25) #object
player2=PlayerCharacter("eion") #object
player3=PlayerCharacter(age=29) #object
player4=PlayerCharacter()

print(player1.name)
print(player1.age)
print(player2.name)
print(player2.age)
print(player3.name)
print(player3.age)
print(player4.name)
print(player4.age)
```

```
lranon
25
eion
95
anonymous
29
anonymous
95
```

```
In [6]: class PlayerCharacter: #class
    playermembership=True #Class Object Attribute
    def __init__(self, age=66, name="anonymous"):
        if (age>=18):
            self.name=name #attribute
            self.age=age
    def run(self): # run is a class method
        return(f"My name is {self.name}")
player1=PlayerCharacter(25,"lranon") #object
player2=PlayerCharacter(name="eion") #object
player3=PlayerCharacter(29) #object
player4=PlayerCharacter()

print(player1.name)
print(player1.age)
print(player2.name)
print(player2.age)
print(player3.name)
print(player3.age)
print(player4.name)
print(player4.age)
```

```
lranon
25
eion
66
anonymous
29
anonymous
66
```

```
In [5]: class PlayerCharacter:
    playermembership=True
    def __init__(self, age=17, name="anonymous"):
        if (age>=18):
            self.name=name
            self.age=age
    def run(self):
        return(f"My name is {self.name}")
player1=PlayerCharacter(25,"lranon")
player3=PlayerCharacter(29)
player2=PlayerCharacter(name="eion")
player4=PlayerCharacter()

print(player1.name)
```

```

print(player1.age)
print(player3.name)
print(player3.age)
print(player2.name)
#for player 2, we didnt give argument, so it takes the default age parameter which is 17, but as there is age restriction
#default age=17, but attributes can be accessed only if age>18, hence the attributes are not accessible and throws error.
print(player2.age)
print(player4.name)
print(player4.age)

```

```

lranon
25
anonymous
29

```

```

-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-5-a17273857d10> in <module>
      16     print(player3.name)
      17     print(player3.age)
--> 18     print(player2.name)#for player 2, we didnt give argument, so it takes the default age parameter which is 0, but as there is age restriction
      19                                         #default age=0, but attributes can be accessed only if age>18, hence the attributes are not accessible and throws error.
      20     print(player2.age)

AttributeError: 'PlayerCharacter' object has no attribute 'name'

```

```

In [42]: class PlayerCharacter:
    playermembership=True
    def __init__(self,age=19,name="anonymous"):
        if (age>=18):
            self.name=name
            self.age=age
    def run(self):
        return(f"My name is {self.name}")
player1=PlayerCharacter(25,"lranon")
player3=PlayerCharacter(29)
player2=PlayerCharacter(name="eion")
player4=PlayerCharacter()

print(player1.name)
print(player1.age)
print(player3.name)
print(player3.age)
print(player2.name)
print(player2.age)
print(player4.name)
print(player4.age)

```

```

lranon
25
anonymous
29
eion
19
anonymous
19

```

```

In [1]: class PlayerCharacter:
    playermembership=True
    def __init__(self,age=19,name="anonymous"):
        if (age>=18):
            self.name=name
            self.age=age
    def run(self):
        return(f"My name is {self.name}")
player1=PlayerCharacter(25,"lranon")
player3=PlayerCharacter(29)
player2=PlayerCharacter(name="eion")
# player4=PlayerCharacter()

print(player1.name)
print(player1.age)
print(player3.name)
print(player3.age)
print(player2.name)
print(player2.age)
print(player4.name)
print(player4.age)

```

```

lranon
25
anonymous
29
eion
19

```

```
NameError Traceback (most recent call last)
<ipython-input-1-146dda12e0c7> in <module>
    18     print(player2.name)
    19     print(player2.age)
--> 20     print(player4.name)
    21     print(player4.age)

NameError: name 'player4' is not defined
```

Excise: Cats Everywhere:

```
In [6]: #Given the below class:
class Cat:
    species = 'mammal'
    def __init__(self, name, age):
        self.name = name
        self.age = age
# 1 Instantiate the Cat class with 3 cats
# 2 Create a function that finds the oldest cat
# 3 Print out: "The oldest cat is x years old.". x will be the oldest cat age by using the function in #2
```

```
In [5]: class Cat:
    species = 'mammal'
    def __init__(self, name, age):
        self.name = name
        self.age = age

# 1 Instantiate the Cat class with 3 cats
cat1 = Cat("Peanut", 3)
cat2 = Cat("Garfield", 5)
cat3 = Cat("Snickers", 1)
print(cat3.name)

# 2 Create a function that finds the oldest cat
def oldest_cat(*args):
    return max(args)

# 3 Print out: "The oldest cat is x years old.". x will be the oldest cat age by using the function in #2
# x=oldest_cat(cat1.age, cat2.age, cat3.age)
# print(f"The oldest cat is {x} years old.")
print(f"The oldest cat is {oldest_cat(cat1.age, cat2.age, cat3.age)} years old.")
```

Snickers

The oldest cat is 5 years old.

```
In [7]: class Cat:
    species = 'mammal'
    def __init__(self, name, age):
        self.name = name
        self.age = age
cat1 = Cat("Peanut", 3)
cat2 = Cat("Garfield", 5)
cat3 = Cat("Snickers", 1)
def oldest_cat(*args):
    return max(args)
oldcat=oldest_cat(cat1.age, cat2.age, cat3.age)
if oldcat is 3:
    print(f"The oldest cat is Peanut and it is {oldcat} years old.")
elif oldcat is 5:
    print(f"The oldest cat is Garfield and it is {oldcat} years old.")
else:
    print(f"The oldest cat is Snickers and it is {oldcat} years old.)
```

The oldest cat is Garfield and it is 5 years old.

Duck Typing:

- <https://www.geeksforgeeks.org/duck-typing-in-python/>
- <https://realpython.com/duck-typing-python/>
- <https://towardsdatascience.com/duck-typing-python-7aeac97e11f8>
- <https://stackoverflow.com/questions/4205130/what-is-duck-typing>
- <https://www.youtube.com/watch?v=mbpHqb04BI8>

Duck typing is a programming concept in Python that allows you to check if an object can be used in a certain way, rather than checking its type. If an object looks like a duck, swims like a duck, and quacks like a duck, then it is considered a duck – regardless of what its actual type is. This concept is based on the duck typing principle which states that the type or the class of an object should not be explicitly checked to verify if it can perform a certain action, but instead, it should be tested directly if it can perform that action. Here's an example of how duck typing can be implemented in Python:

```
def add(a, b):
    """
    Function to add two numbers.
    This function uses duck typing to add two numbers, regardless of their type.
    If the objects can be added together, they are considered numbers.
    """
```

```

try:
    result = a + b # Try to add the two objects
    return result # Return the result
except TypeError: # If a TypeError occurs,
    print("The provided objects cannot be added.") # print an error message

# Test the function with different data types
print(add(3, 5))      # Integers
print(add(4.2, 3.1))   # Floats
print(add("Hello, ", "World!")) # Strings

```

In this example, the `add` function does not check the types of `a` and `b`. Instead, it tries to add them together. If the objects can be added, the addition will succeed. If not, a `TypeError` will be raised, and the function will print an error message. This way, the function uses duck typing to determine if the provided objects can be added, without explicitly checking their types. */

@classmethod and @staticmethod

<https://www.makeuseof.com/tag/python-instance-static-class-methods/>

1. We generally use class method to create factory methods. Factory methods return class object (similar to a constructor) for different use cases.
2. We generally use static methods to create utility functions.

We learned that we were able to create an actual attribute (class object attribute) for the class. But what about a method. Is there a way to do something like what we do with attributes but for methods (i.e., method for the class) well there is and we use a decorator

```

In [32]: class PlayerCharacter: #class
    playermembership=True # actual attribute for the class
    def __init__(self, name, age): #constructor method
        self.name=name #attributes
        self.age=age

    def run(self): #method
        return(f"My name is {self.name}")

    @classmethod #decorator
    # method for the class
    def add_things(num1, num2):
        return num1+num2

player1=PlayerCharacter("lan",25) #object
print(player1.add_things) #player1 has access to add_things method
print(player1.add_things(2,3))

<bound method PlayerCharacter.add_things of <class '__main__.PlayerCharacter'>>
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-32-a9d6c1222337> in <module>
      15 player1=PlayerCharacter("lan",25) #object
      16 print(player1.add_things) #player1 has access to add_things method
---> 17 print(player1.add_things(2,3))

TypeError: add_things() takes 2 positional arguments but 3 were given

```

So when we call `"print(player1.add_things(2,3))"`, we gave it two parameters but then it says that we actually got three and that is because `add_things`,

the `first parameter just like we have self as 1st parameter in run() method`, here the `1st parameter is the cls in classmethod()` and `cls` stands for class.

```

In [6]: # class PlayerCharacter:
#     def __init__(self, name, age):
#         #class state:line 4-5
#         self.name=name
#         self.age=age

```

```

In [39]: class PlayerCharacter: #class
    playermembership=True # actual attribute for the class
    def __init__(self, name, age): #constructor method
        self.name=name #attributes
        self.age=age

    def run(self): #method
        return(f"My name is {self.name}")

    @classmethod #decorator
    # method for the class
    def add_things(cls, num1, num2):
        return num1+num2

player1=PlayerCharacter("lan",25) #object
print(player1.add_things) #player1 has access to add_things method
print(player1.add_things(2,3))

```

```
<bound method PlayerCharacter.add_things of <class '__main__.PlayerCharacter'>>
5

In [38]: class PlayerCharacter: #class
    playermembership=True # actual attribute for the class
    def __init__(self, name, age): #constructor method
        self.name=name #atribues
        self.age=age

    def run(self): #method
        return(f"My name is {self.name}")

    @classmethod #decorator
    # method for the class
    def add_things(hi, num1, num2): #we can give anything as 1st parameter to refer the class but the standard is to use cls.
        return num1+num2
player1=PlayerCharacter("lan",25) #object
print(player1.add_things) #player1 has access to add_things method
print(player1.add_things(2,3))
```

```
<bound method PlayerCharacter.add_things of <class '__main__.PlayerCharacter'>>
5
```

@classmethod :It's a class method - a method on the actual class..

But how is "def add_things(cls,num1,num2):" a class method.

Well it's because we can actually use this without even instantiating a class.

```
In [10]: class PlayerCharacter:
    playermembership=True
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self):
        return(f"My name is {self.name}")

    @classmethod
    def add_things(cls, num1, num2): #can actually use this without even instantiating a class
        return num1+num2

# player1=PlayerCharacter("lan",25)
print(PlayerCharacter.add_things(2,3))
```

```
5
```

```
In [2]: class PlayerCharacter:
    playermembership=True
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self):
        return(f"My name is {self.name}")

    @classmethod
    def add_things(cls, num1, num2):
        return num1+num2

player1=PlayerCharacter("lan",25)
print(player1.add_things(2,3))
```

```
5
```

IMP:

we can use the cls to actually instantiate an object in classmethod

For example we can use the cls to actually instantiate an object in classmethod.

So for example I can say that cls which is the class player character and I'm going to instantiate it remember just like this "player1=PlayerCharacter("lan",25)" with the brackets and with num1+num2 as the second parameter and we'll give it name Teddy if I hit run here. Look at that I've instantiated an object Teddy with the age of what should be 5.

```
In [10]: class PlayerCharacter:
    playermembership=True
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self):
        return(f"My name is {self.name}")

    @classmethod
    def add_things(cls, num1, num2):
        return cls('Teddy', num1+num2) #using the cls to actually instantiate an object in classmethod
        #(name,age) in def __init__(self,name,age) is equivalent to ('Teddy',num1+num2) in return cls('Teddy',num1+num2)
```

```
player3=PlayerCharacter.add_things(2,3)
print(player3.age)
print(player3.name)
```

```
5
Teddy
```

This code defines a class called `PlayerCharacter` in Python. Let's go through it piece by piece:

1. `class PlayerCharacter:` - This line is defining a new class named `PlayerCharacter`.
2. `playermembership=True` - This is a class variable that is set to True. Class variables are shared by all instances of a class. In this case, all player characters have a `playermembership` attribute which is set to True.
3. `def __init__(self, name, age):` - This is the constructor method for the class. It is called when a new instance of the class is created. The `self` parameter is a reference to the instance being initialized, `name` and `age` are other parameters passed to the function.
- 4-5. `self.name=name` and `self.age=age` - These lines are assigning the values passed to the constructor to the current instance of the class.
6. `def run(self):` - This line is defining a new method named `run` for the class. This method takes the instance (`self`) as its first parameter.
7. `return(f"My name is {self.name}")` - This line is returning a string that includes the name attribute of the current instance.
8. `@classmethod` - This decorator allows the method that follows it to be called on the class itself, not just on instances of the class.
9. `def add_things(cls, num1, num2):` - This line is defining a new class method for the class. This method takes the class as its first parameter (`cls`), as well as two additional parameters, `num1` and `num2`.
10. `return cls('Teddy', num1+num2)` - This line is creating a new instance of the class (`cls`) with the name "Teddy" and an age equal to the sum of `num1` and `num2`, then returning this new instance.
11. `player3=PlayerCharacter.add_things(2,3)` - This line is calling the `add_things` class method with the arguments 2 and 3. The method creates a new `PlayerCharacter` instance with the name "Teddy" and an age of 5 (2 + 3), assigns this new instance to the variable `player3`.
- 12-13. `print(player3.age)` and `print(player3.name)` - These lines are printing the age and name attributes of the `player3` instance, respectively.

So, when this code is executed, it will print:

```
5
Teddy
```

Findings:

```
In [2]: class PlayerCharacter:
    membership=True
    def __init__(self, name, age):
        self.name=name
        self.age=age
    def run(self):
        return("Oops")
    def abc(cls,n1,n2):
        return n1 + n2
print(PlayerCharacter.abc(2,5)) #accessing the class method without the decorator
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-2-3f059edb9c19> in <module>
      8     def abc(cls,n1,n2):
      9         return n1 + n2
---> 10 print(PlayerCharacter.abc(2,5)) #accessing the class method without the decorator
      11
TypeError: abc() missing 1 required positional argument: 'n2'
```

```
In [1]: class PlayerCharacter:
    membership=True
    def __init__(self, name, age):
        self.name=name
        self.age=age
    def run(self):
        return("Oops")
    def abc(cls,n1,n2):
        return n1 + n2
print(PlayerCharacter.abc(2,5,3)) #accessing the class method without the decorator
```

```
8
```

```
In [3]: class PlayerCharacter:
    membership=True
    def __init__(self, name, age):
        self.name=name
        self.age=age
    def run(self):
```

```

        return("Oops")
    @classmethod
    def abc(cls,n1,n2):
        return n1 + n2
print(PlayerCharacter.abc(2,5))

```

7

```
In [1]: class PlayerCharacter:
    membership=True
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def run(self):
        return("Oops")
    def abc(n1,n2): #No cls
        return n1 + n2
print(PlayerCharacter.abc(2,5))#accessing the class method without the decorator
```

7

```
In [5]: class PlayerCharacter:
    membership=True
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def run(self):
        return("Oops")
    def abc(n1,n2):#no cls
        return n1 + n2
play2=PlayerCharacter("candy",55)
print(play2.abc(1,2))
```

```

-----  

TypeError                                     Traceback (most recent call last)  

<ipython-input-5-3a0475002cc9> in <module>  

      9     return n1 + n2  

     10 play2=PlayerCharacter("candy",55)  

--> 11 print(play2.abc(1,2))

TypeError: abc() takes 2 positional arguments but 3 were given

```

```
In [11]: class PlayerCharacter:
    membership=True
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def run(self):
        return("Oops")
    def abc(cls,n1,n2):
        return n1 + n2
play2=PlayerCharacter("candy",55)
print(play2.abc(1,2))
```

3

```
In [7]: class PlayerCharacter:
    membership=True
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def run(self):
        return("Oops")
    @classmethod
    def abc(cls,n1,n2):
        return n1 + n2
play2=PlayerCharacter("candy",55)
print(play2.abc(1,2))
```

3

@staticmethod:

1. A class method takes `cls` as first parameter while a static method needs no specific parameters.
2. A class method can access or modify class state while a static method can't access or modify it.
3. In general, static methods know nothing about class state. ... On the other hand class methods must have class as parameter.

`Staticmethod` works the exact same way as `classmethod` except you do not have access to this `cls` or the class. so you can't do something like we did above.

Instead we just perform some sort of method like `add_things` and the only difference between the two is the idea that we don't have access in our parameters to this `cls` in `staticmethod`. so we would use something like static method where we don't care anything about the class state,a class state is something like these "`self.name=name, self.age=age`", we don't care about the attributes really. we use something like a class method when we do care about the attributes and maybe we want to modify them or change them.

```
In [1]: class PlayerCharacter:
    playermembership=True
    def __init__(self,name,age):
        self.name=name
```

```

        self.age=age

    def run(self):
        return(f"My name is {self.name}")

    @staticmethod
    def add_things2(num1,num2):
        return num1+num2

#Accessing @staticmethod via Class Instances
player3=PlayerCharacter("joe",23)
print(player3.add_things2(1,2))

```

3

```
In [9]: class PlayerCharacter:
    playermembership=True
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def run(self):
        return(f"My name is {self.name}")

    @staticmethod
    def add_things2(num1,num2): #can use this without instantiating a class
        return num1+num2

print(PlayerCharacter.add_things2(5,8))
```

13

```
In [12]: class PlayerCharacter:
    playermembership=True
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def run(self):
        return(f"My name is {self.name}")

    @classmethod
    def add_things0(cls,num1,num2):
        return num1+num2

    @classmethod
    def add_things(cls,num1,num2):
        return cls('Teddy',num1+num2)
    #(name,age) in def __init__(self,name,age) == ('Teddy',num1+num2) in return cls('Teddy',num1+num2)

    @staticmethod
    def add_things2(num1,num2):
        return num1+num2

player1=PlayerCharacter("lan",25)
print(player1.name)
print(player1.age)
print(PlayerCharacter.add_things0(20,20))
print(player1.add_things0(20,25))

player3=PlayerCharacter.add_things(2,20)
print(player3.age)

print(PlayerCharacter.add_things2(5,8))

player4=PlayerCharacter("joe",43)
print(player4.name)
print(player4.age)
print(player4.add_things2(1,2))
```

```

lan
25
40
45
22
13
joe
43
3

```

Instance methods can only be called from a class instance, it cannot be called via the Class

```
In [7]: class PlayerCharacter:
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def run(self): #instance method
        return(f"My name is {self.name}")
player1=PlayerCharacter("lan",25)
print(player1.run())
```

```
My name is lan
```

```
In [8]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self): #instance method
        return(f"My name is {self.name}")
print(PlayerCharacter.run())
```

Instance methods can only be called from a class instance, it cannot be called via the Class

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-8-4373af15e12e> in <module>
      6     def run(self): #instance method
      7         return(f"My name is {self.name}")
----> 8 print(PlayerCharacter.run())

TypeError: run() missing 1 required positional argument: 'self'
```

@classmethod: Class methods can be called from both a class instance as well as from a Class

```
In [4]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    @classmethod
    def add_things(cls, num1, num2):
        return num1+num2

print(PlayerCharacter.add_things(2,20))
```

22

```
In [5]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    @classmethod
    def add_things(cls, num1, num2):
        return num1+num2

player1=PlayerCharacter("lan",25)
print(player1.add_things(30,20))
```

50

@staticmethod can be called from both a class instance as well as from a Class.

```
In [2]: class PlayerCharacter:
    playermembership=True
    def __init__(self, name, age):
        self.name=name
        self.age=age
    @staticmethod
    def add_things2(num1, num2):
        return num1+num2

player4=PlayerCharacter("joe",43)
print(player4.add_things2(9,10))
```

19

```
In [10]: class PlayerCharacter:
    playermembership=True
    def __init__(self, name, age):
        self.name=name
        self.age=age
    @staticmethod
    def add_things2(num1, num2):
        return num1+num2

print(PlayerCharacter.add_things2(1,2))
```

3

Instance Methods, Class Methods, Static Methods:

<https://www.youtube.com/watch?v=lVfGQOzzRCM>

Instance methods are used to work with the instance variables

The instance methods are of two types:

- Accessor methods - used to just fetch the values (instance variables)

- Mutator methods - used to modify the values (instance variables)

```
In [1]: class Student:
    school = 'Minerva' #class/static variable
    def __init__(self,m1,m2,m3):
        self.m1=m1 #instance variables
        self.m2=m2
        self.m3=m3

    def avg(self): #instance method
        return (self.m1+self.m2+self.m3)/3

    def get_m1(self): #Accessor methods
        return self.m1

    def set_m1(self): #Mutator methods
        return self.m1

s1 = Student(12,45,78)
s2 = Student(78,89,75)

print(s1.avg())
45.0
```

Class methods are used to work with the class variables

```
In [2]: class Student:
    school = 'Minerva' #class/static variable
    def __init__(self,m1,m2,m3):
        self.m1=m1 #instance variables
        self.m2=m2
        self.m3=m3

    def avg(self): #instance method
        return (self.m1+self.m2+self.m3)/3

    def info(cls):
        return cls.school

s1 = Student(12,45,78)
s2 = Student(78,89,75)

print(s1.avg())
print(Student.info())
45.0
```

```
-----
TypeError                                     Traceback (most recent call last)
Cell In[2], line 19
  16 s2 = Student(78,89,75)
  17 print(s1.avg())
--> 19 print(Student.info())

TypeError: Student.info() missing 1 required positional argument: 'cls'
```

```
In [3]: class Student:
    school = 'Minerva' #class/static variable
    def __init__(self,m1,m2,m3):
        self.m1=m1 #instance variables
        self.m2=m2
        self.m3=m3

    def avg(self): #instance method
        return (self.m1+self.m2+self.m3)/3

    @classmethod
    def getschool(cls):
        return cls.school

s1 = Student(12,45,78)
s2 = Student(78,89,75)

print(s1.avg())
print(Student.getschool())
45.0
Minerva
```

Static Method - This Method has nothing to do with the instance variables, nothing to do with the class variables

- Used to perform any operations related to the variables of other objects/other classes or some generic operations which are not related to the class

```
In [4]: class Student:
    school = 'Minerva' #class/static variable
```

```

def __init__(self,m1,m2,m3):
    self.m1=m1 #instance variables
    self.m2=m2
    self.m3=m3

def avg(self): #instance method
    return (self.m1+self.m2+self.m3)/3

@classmethod
def getschool(cls):
    return cls.school

def info(): #no self, no cls
    return 'This is the static method'

s1 = Student(12,45,78)
s2 = Student(78,89,75)

print(s1.avg())
print(Student.getschool())
print(Student.info())

```

45.0
Minerva
This is the static method

```

In [5]: class Student:
    school = 'Minerva' #class/static variable
    def __init__(self,m1,m2,m3):
        self.m1=m1 #instance variables
        self.m2=m2
        self.m3=m3

    def avg(self): #instance method
        return (self.m1+self.m2+self.m3)/3

    @classmethod
    def getschool(cls):
        return cls.school

    @staticmethod
    def info(): #no self, no cls
        return 'This is the static method'

s1 = Student(12,45,78)
s2 = Student(78,89,75)

print(s1.avg())
print(Student.getschool())
print(Student.info())

```

45.0
Minerva
This is the static method

In Python, the `@staticmethod` decorator is used to define a method within a class that does not access or modify the class or instance state. It is similar to a regular function but is defined within a class for organizational purposes. Here's how it works and why it's useful:

1. **Organization:** Placing related functions within a class can help in organizing code logically, especially if those functions are closely associated with the class but do not need access to instance or class variables.
2. **Code Clarity:** It makes it clear that the method does not depend on instance or class state, which can improve code readability and maintainability.
3. **Namespacing:** By defining a method within a class, you're effectively namespacing it under that class. This can help avoid naming conflicts, especially in larger codebases.
4. **Utility Functions:** It's useful for defining utility functions that are closely related to the class but don't require access to instance variables. These functions can be accessed directly through the class without needing an instance.

Here's an example to illustrate its usage:

```

class MathUtils:
    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def multiply(x, y):
        return x * y

# Using static methods without creating an instance of MathUtils
print(MathUtils.add(5, 3)) # Output: 8
print(MathUtils.multiply(5, 3)) # Output: 15

```

In the example above, `add()` and `multiply()` are static methods. They don't depend on any instance variables or methods of `MathUtils`. You can call them directly on the class `MathUtils` without creating an instance of it.

In Python, you can define regular methods within a class without using the `@staticmethod` decorator, and they can still be called without creating an instance of the class. Here's the same example without using `@staticmethod`:

```
class MathUtils:
    def add(x, y):
        return x + y

    def multiply(x, y):
        return x * y

# Using methods without creating an instance of MathUtils
print(MathUtils.add(5, 3)) # Output: 8
print(MathUtils.multiply(5, 3)) # Output: 15
```

In this case, the methods `add()` and `multiply()` can still be called directly on the class `MathUtils`, just like with static methods. However, there's a subtle difference:

1. **Decorated with `@staticmethod`**: Using `@staticmethod` explicitly communicates the intention that these methods do not depend on the instance or class state. It's a way of documenting that these methods are self-contained and do not need access to instance variables or methods.
2. **Without `@staticmethod`**: While the methods can still be called without creating an instance, not using `@staticmethod` may imply to readers that these methods might depend on instance or class variables, especially in more complex classes.

In summary, both approaches achieve the same result, but using `@staticmethod` can provide clearer intent and documentation in your code. It explicitly marks the method as independent of instance or class state.

In Python, a class method is a method which is bound to the class and not an instance of a class. It can be called on a class or an instance of a class. Class methods are created using the `@classmethod` decorator. They are especially useful when you want to create utility methods that don't depend on the state of an instance.

Here is an example of a class method:

```
class MyClass:
    counter = 0

    @classmethod
    def update_counter(cls, increment):
        cls.counter += increment

    @classmethod
    def get_counter(cls):
        return cls.counter
```

In this example, `update_counter` and `get_counter` are class methods. They manipulate and retrieve the value of `counter`, a variable associated with the class itself, not an instance of the class.

We can call these methods on the class or an instance of the class:

```
# Call class methods on the class
MyClass.update_counter(5)
print(MyClass.get_counter()) # Output: 5

# Call class methods on an instance
instance = MyClass()
instance.update_counter(3)
print(instance.get_counter()) # Output: 8
```

As you can see, calling `update_counter` and `get_counter` on an instance works, but it's generally better to call them on the class because it makes the code clearer.

Note: Class methods can also access the instance with `cls` (the first argument), for example:

```
class MyClass:
    @classmethod
    def my_class_method(cls, instance):
        print(f"This is a class method of {cls.__name__} and I can access the instance: {instance}")

my_instance = MyClass()
MyClass.my_class_method(my_instance)
```

In this case, the instance is passed as an argument, so it can be used within the class method.

```

1 class MyClass:
2     @classmethod
3     def my_class_method(cls, instance):
4         print(f"This is a class method of {cls.__name__} and I can access the instance: {instance}")
5 MyClass.my_class_method(12)

```

This is a class method of MyClass and I can access the instance: 12

```

1 class MyClass:
2     @classmethod
3     def my_class_method(cls, instance):
4         print(f"This is a class method of {cls.__name__} and I can access the instance: {instance}")
5 my_instance=MyClass()
6 MyClass.my_class_method(my_instance)

```

This is a class method of MyClass and I can access the instance: <__main__.MyClass object at 0x000001ED9DD847C0>

In Python, a static method is a method that belongs to a class rather than an instance of a class. It can be called on an instance or on a class. In other words, static methods are shared by all instances of a class and are independent of the state of the instance.

To define a static method, you use the `@staticmethod` decorator. Here's an example:

```

class MyCalculator:
    def __init__(self, value=0):
        self.value = value

    @staticmethod
    def static_add(a, b):
        return a + b

```

In this example, `static_add` is a static method because it's decorated with `@staticmethod`. This means it can be called both on an instance of `MyCalculator` and on `MyCalculator` itself:

```

calc = MyCalculator()

# Call the static method on the instance
result1 = calc.static_add(5, 10)
print(result1) # Output: 15

# Call the static method on the class
result2 = MyCalculator.static_add(15, 20)
print(result2) # Output: 35

```

As you can see, the static method doesn't have access to the instance (like `self`) or the class (like `cls`) when it's called. It only has access to its own arguments.

Static methods are useful when you want a method to belong to a class rather than an instance, and you don't need to access the instance or class state. They're commonly used for utility methods that perform a task in isolation.

In Python, both `classmethod` and `staticmethod` are used to bind methods to classes, but they have different use-cases.

`classmethod` is used when you want to create a method that is bound to the class and can access class-level attributes or methods, but doesn't necessarily require an instance of the class. It's commonly used for utility methods that perform some operation on the class but don't depend on the state of an instance.

For example, a `classmethod` could be used to calculate the number of instances of a class that have been created so far.

```

class MyClass:
    instances = 0

    def __init__(self):
        MyClass.instances += 1

    @classmethod
    def get_instance_count(cls):
        return cls.instances

```

`staticmethod` is used when you want to create a method that doesn't depend on either the class or an instance. It's commonly used for simple utility functions that perform a task independent of the class or instance.

For example, a `staticmethod` could be used to convert a temperature from Celsius to Fahrenheit.

```

class MyClass:
    @staticmethod
    def celsius_to_fahrenheit(celsius):
        return celsius * 9/5 + 32

```

In summary, use `classmethod` when you need to access class-level attributes or methods, and `staticmethod` when you just need a simple utility function that doesn't depend on the class or an instance.

In Python, both static methods and class methods are used to create methods that aren't bound to an instance of a class. However, they differ in how they are called and what they can access within the class.

Static Method: A static method is a method that belongs to a class rather than an instance of a class. It can be called on an instance or on a class. It doesn't have access to self or the class instance (except as explicit arguments) and doesn't allow modification of the class or instance.

Class Method: A class method is a method that belongs to a class and not an instance of a class. It can be called on an instance or on a class. It has access to the class and its attributes, but not the instance. Class methods are created using the @classmethod decorator. They are especially useful for creating utility methods that don't depend on the state of an instance.

Here is an example to illustrate the difference:

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

    @staticmethod
    def static_method():
        print("This is a static method")

    @classmethod
    def class_method(cls):
        print("This is a class method")

    def instance_method(self):
        print(f"This is an instance method, accessed by {self.name}")

# Creating an instance of AiDE
aide = AiDE("AiDE instance")

# Calling the static method
aide.static_method()
# Output: This is a static method

AiDE.static_method()
# Output: This is a static method

# Calling the class method
aide.class_method()
# Output: This is a class method

AiDE.class_method()
# Output: This is a class method

# Calling the instance method
aide.instance_method()
# Output: This is an instance method, accessed by AiDE instance
```

In this code, `static_method()` is a static method and `class_method()` is a class method. As shown, both can be called on an instance or on a class. The instance method `instance_method()` can access the instance (`self.name` in this case), while both static and class methods cannot access the instance.

Review of OOPs:

- Instance methods need a class instance and can access the instance through `self`. It requires an object of its class to be created before it can be called.
- Class methods don't need a class instance. They can't access the instance (`self`) but they have access to the class itself via `cls`.
- Static methods don't have access to `cls` or `self`.

```
class NameOfClass:
    classattribute=value
    def __init__(self, param1, param2):
        self.param1=param1
        self.param2=param2
    def method(self): #instance method
        #code

    @classmethod
    def cls_method(cls, param1, param1):
        #code

    @staticmethod
    def stc_method(param1, param1):
        #code
```

In [5]: `class PlayerCharacter:
 def __init__(self, name, age): #self refers to the PlayerCharacter class`

```

        self.name=name
        self.age=age
    def run(self):
        return self
player1=PlayerCharacter("lranon",25)
print(player1.run)

<bound method PlayerCharacter.run of <__main__.PlayerCharacter object at 0x000000F16CE75588>>

In [2]: class PlayerCharacter:
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def run(self,num1,num2): #instance method
        return num1+num2
player1=PlayerCharacter("lan",25)
print(player1.run(4,5))

9

```

Data Class:

Data class is a decorator in Python which helps to create classes automatically. It is a convenient way to create simple classes that have a few fields and often just need basic object functionality.

With Python 3.7+, you can use the `@dataclass` decorator from the `dataclasses` module to achieve this. This decorator adds a few useful methods to the class:

- `__init__(self, ...)` : This method is the class constructor which initializes the attributes.
- `__repr__(self)` : This method returns a string that describes the instance of the class (the object).
- `__eq__(self, other)` : This method checks if two instances of the class are equal (have the same attributes).
- `__hash__(self)` : This method returns a hash of the object, which can be used for operations like dictionary keys.

Here's an example of how to use a data class in Python:

```

from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int

p = Point(1, 2)
print(p) # Output: Point(x=1, y=2)

```

In this example, `Point` is a data class with two fields `x` and `y`, both of type `int`. The `dataclass` decorator generates the necessary code for these fields, including the methods mentioned above.

Sure, here's an example of how you can use a data class in Python:

```

from dataclasses import dataclass

@dataclass
class User:
    name: str
    age: int
    is_active: bool = False

# Creating an instance of User
user1 = User("John Doe", 25)

# Accessing attributes
print(user1.name)      # Output: John Doe
print(user1.age)        # Output: 25
print(user1.is_active) # Output: False

# You can also update the attributes
user1.name = "Jane Doe"
user1.age = 30
user1.is_active = True

print(user1.name)      # Output: Jane Doe
print(user1.age)        # Output: 30
print(user1.is_active) # Output: True

```

This is a simple example of a `User` data class with three attributes: `name`, `age`, and `is_active`. The `is_active` attribute has a default value of `False`. We then create an instance of the `User` class, access its attributes, and update them.

4 pillars of OOP: Encapsulation, Abstraction, Inheritance, Polymorphism

Encapsulation:

What is encapsulation? Encapsulation is the binding of data and functions that manipulate that data and we encapsulate into one big box so that we keep everything in this box that users or code or other machines can interact with. This data and functions are what we call attributes and methods.

We're able to encapsulate the functionality over player character by having name and age data or attributes and also have functions that can act upon this name and age.

A class is an example of encapsulation as it encapsulates all the data that is functions, variables, etc.

```
In [3]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self, num1, num2):
        return num1+num2
player1=PlayerCharacter("lan",25)
print(player1.run(4,5))
```

9

Abstraction:

Abstraction means hiding of information or abstracting away information and giving access to only what's necessary. So whatever the user or the programmer or the machine is interested in that's the only thing we give access to.

```
In [13]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self):
        return 'run'

    def fun(self):
        return f'My name is {self.name}, I am {self.age} years old.'

player1=PlayerCharacter("Ian",25)
print(player1.fun())
```

My name is Ian, I am 25 years old.

Now abstraction can actually be seen here right.

When I do player1.fun() you're seeing abstraction in action because when I click Run I get this string but when I call fun() by doing player1.fun() I don't really care how fun() is implemented. All I know is that player1 has access to the fun() method and I can use it.

```
In [14]: tuple=(1,2,1,3,1)
print(tuple.count(1)) # i can count how many times a value is present in the tuple by simply writing ".count(value)"
```

3

Now do I need to know how the count method was implemented to do `print(tuple.count(1))` - No.

This is called Abstraction, the actual implementation of count() method is hidden but we have access to the count()method and we can use it when required.

```
In [3]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self):
        return 'run'

    def fun(self):
        return f'My name is {self.name}, I am {self.age} years old.'

player1=PlayerCharacter("Ian",25)
player1.name="!!!"
player1.fun='hello'
print(player1.name)
print(player1.fun)
```

!!!
hello

Abstraction is good, but hold on a second here this is bad if I have a class that I've abstracted away but anybody can come along.

Any programmer can come along and just remove all my hard work and overwrite it like that, like i had modified the name and fun. Thats very bad.

Public Vs Private Variables:

I mean sure if we instantiate a new object we will have our run and speak but Player1 now is completely useless as we had modified the values.
<https://www.geeksforgeeks.org/encapsulation-in-python/>

```
In [6]: class PlayerCharacter:
    def __init__(self, name, age):
        self.name=name
        self.age=age

    def run(self):
        return 'run'

    def fun(self):
        return f'My name is {self.name}, I am {self.age} years old.'

player1=PlayerCharacter("Ian",25)
player1.name="!!!" # here i am modifying the value
player1.fun='hello' # here i am modifying fun into string value instead of a function
print(player1.name)
print(player1.fun)

player2=PlayerCharacter("Ryan",55)
print(player2.name)
print(player2.fun())

!!!
hello
Ryan
My name is Ryan, I am 55 years old.
```

The idea behind abstraction is that we hide away information and only give access to things that a user is concerned about.

Some languages allow us to have private variables for example in a language like Java. I can actually say that an attribute is private and I won't be able to access it and modify it.

To accomplish this in Python, just follow the convention by prefixing the name of the variable by a single underscore "_".

Does this give any special power - No.

This is just a convention that is as Python programmers. We determined that hey if we see underscore in our code that most likely means that this should be a private variable

```
In [10]: class PlayerCharacter:
    def __init__(self, name, age):
        self._name=name
        self._age=age

    def run(self):
        return 'run'

    def fun(self):
        return f'My name is {self._name}, I am {self._age} years old.'

player2=PlayerCharacter("Ryan",55)
print(player2._name)
print(player2.fun())

Ryan
My name is Ryan, I am 55 years old.
```

We can see that even after declaring the name and age as private variables, I am able to modify them. I click Run this still works.

That's because like I said there's no true private variables but as programmers we've decided that underscore means that you shouldn't touch us.

If you see underscore this shouldn't be modified. This should be private and that's how we achieve privacy in Python. We all kind of agreed to say hey let's keep this little thing private. So if you ever want to keep a method or an attribute private you just put an underscore in front of it. But it's no guarantee.

```
In [11]: class PlayerCharacter:
    def __init__(self, name, age):
        self._name=name
        self._age=age

    def run(self):
        return 'run'

    def fun(self):
        return f'My name is {self._name}, I am {self._age} years old.'

player1=PlayerCharacter("Ian",25)
player1.name="!!!" # here i am modifying the value
player1.fun='hello' # here i am modifying fun into string value instead of a function
print(player1.name)
print(player1.fun)

player2=PlayerCharacter("Ryan",55)
```

```

print(player2._name)
print(player2.fun())
!!!
hello
Ryan
My name is Ryan, I am 55 years old.

```

What about this double underscore init. Again this is something we're going to be speaking about very shortly but it's called a Dunder method that is it's built into python and we usually never write our own Dunder methods. So once again this double underscore is also a convention to let people know you shouldn't really touch this or modify this with each data type.

Inheritance:

https://www.w3schools.com/python/python_inheritance.asp

Inheritance allows new objects to take on the properties of existing objects. So you can inherit classes.

1. Parentheses are optional for class definitions where the class does not inherit from a parent class.
2. Parentheses are mandatory when doing inheritance.

```

class Student(Person):

```

Inheritance allows us to define a class that inherits all the methods and properties from another class.

1. Parent class is the class being inherited from, also called base class.
2. Child class is the class that inherits from another class, also called derived class.

Create a Parent Class:

```

In [4]: # Any class can be a parent class, so the syntax is the same as creating any other class:
# Create a class named Person, with firstname and lastname properties, and a printname method:

class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        return(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:
x = Person("John", "Doe")
x.printname()

Out[4]: ('John', 'Doe')

```

Create a Child Class:

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class

```

In [16]: # Create a class named Student, which will inherit the properties and methods from the Person class:
class Student(Person):
    pass

# Note: Use the pass keyword when you do not want to add any other properties or methods to the class.
# Now the Student class has the same properties and methods as the Person class.

```

```

In [4]: # Use the Student class to create an object, and then execute the printname method:
x = Student("Mike", "Olsen")
x.printname()

```

Mike Olsen

Add the `__init__()` function to the Student class:

Add the `__init__()` Function. So far we have created a child class that inherits the properties and methods from its parent.

We want to add the `__init__()` function to the child class (instead of the pass keyword).

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

```

In [2]: # class Student(Person):
#     def __init__(self, fname, lname):
#         #add properties etc.

```

IMP:

1. When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

Note: The child's `__init__()` function overrides the inheritance of the parent's `__init__()` function.

2. To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function

```
In [11]: class Person:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

x = Person("kij", "hik")
print(x.fname)

class Student(Person):
    def __init__(self, fname, lname): #in the __init__ of child class, the attributes of parent class also needs to be included
        pass
y = Student("ki", "hik")
print(y.fname)
```

kij

```
-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-11-cd9bd348ead1> in <module>
      11         pass
      12 y = Student("ki", "hik")
----> 13 print(y.fname)

AttributeError: 'Student' object has no attribute 'fname'
```

```
In [20]: class Student(Person):
    def __init__(self, fname, lname):
        Person.__init__(self, fname, lname)
```

Now we have successfully added the `__init__()` function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

IMP:

- While defining the `__init__()` of child class, we need to pass all the parameters of `__init__()` of parent class also in the `__init__()` of child class

Use the super() Function:

Note: `self` not required

By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

```
In [9]: # Python also has a super() function that will make the child class inherit all the methods and properties from its parent:
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname) #self not required
```

Add Properties:

```
In [10]: # Add a property called graduationyear to the Student class:
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
        self.graduationyear = 2019
```

```
In [3]: #In the example below, the year 2019 should be a variable, and passed into the Student class when creating student objects.
# To do so, add another parameter in the __init__() function:
```

```
In [12]: # Add a year parameter, and pass the correct year when creating objects:

class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

Add Methods:

```
In [ ]: # Add a method called welcome to the Student class:

class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year
```

```
def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

Note: If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

```
In [18]: class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
x = Person("John", "Doe")
x.printname()

class Student(Person):
    pass
x = Student("Mike", "Olsen")
x.printname()
```

John Doe

Mike Olsen

```
In [26]: class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
x = Person("John", "Doe")
x.printname()

class Student(Person):
    def __init__(self, fname, lname):
        pass
x = Student("Mike", "Olsen")
x.printname()

# When you add the init() function, the child class will no longer inherit the parent's init() function.
# Note: The child's init() function overrides the inheritance of the parent's init() function.
# To keep the inheritance of the parent's init() function, add a call to the parent's init() function
```

John Doe

```
-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-26-96867d9280cb> in <module>
      13     pass
      14 x = Student("Mike", "Olsen")
--> 15 x.printname()

<ipython-input-26-96867d9280cb> in printname(self)
      5
      6     def printname(self):
--> 7         print(self.firstname, self.lastname)
      8 x = Person("John", "Doe")
      9 x.printname()

AttributeError: 'Student' object has no attribute 'firstname'
```

```
In [37]: class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
x = Person("John", "Doe")
x.printname()

class Student(Person):
    def __init__(self, fname, lname):
        Person.__init__(self, fname, lname)
        #To keep the inheritance of the parent's init() function, add a call to the parent's init() function
x = Student("Mike", "Olsen")
x.printname()
```

John Doe

Mike Olsen

super()

while using super(), No need to give self as parameter

```
In [38]: class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname
```

```

def printname(self):
    print(self.firstname, self.lastname)
x = Person("John", "Doe")
x.printname()

class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname) # No need to give self as parameter
#By using the super() function, you do not have to use the name of the parent element,
#it will automatically inherit the methods and properties from its parent
x = Student("Mike", "joe")
x.printname()

```

John Doe
Mike joe

Add a property called graduationyear to the Student class:

```
In [42]: class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
x = Person("John", "Doe")
x.printname()

class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
        self.graduationyear = 2019
x = Student("Mike", "joe")
x.printname()
```

John Doe
Mike joe

In the example above, the year 2019 should be a variable, and passed into the Student class when creating student objects. To do so, add another parameter in the `__init__()` of child class:

Add a year parameter, and pass the correct year when creating objects:

```
In [44]: class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
x = Person("John", "Doe")
x.printname()

class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
x.printname()
```

John Doe
Mike Olsen

Add Methods:

```
In [51]: class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
x = Person("John", "Doe")
x.printname()

class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year
    def welcome(self):
        print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
y = Student("Mike", "Olsen", 2019)
y.printname()
y.welcome()
```

John Doe
Mike Olsen
Welcome Mike Olsen to the class of 2019

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

```
In [5]: class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
x = Person("John", "Doe")
x.printname()

class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year
    def printname(self):
        print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
y = Student("Mike", "Olsen", 2019)
print(y.graduationyear)
y.printname()
```

John Doe
2019
Welcome Mike Olsen to the class of 2019

Note: self.firstname=fname, after instantiating the object we should write obj.firstname to access the fname

```
In [12]: class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
x = Person("John", "Doe")
x.printname()
print(x.firstname)

class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year
    def printname(self):
        print(f"Welcome {self.firstname} {self.lastname} to the class of {self.graduationyear}")
y = Student("Mike", "Olsen", 2019)
print(y.graduationyear)
y.printname()
```

John Doe
John
2019
Welcome Mike Olsen to the class of 2019

Classname.__init__(self,attribute1,attribute2) &
super().__init__(attribute1,attribute2)

We can call method of parent class in child class using:

- Person.testfn(self) or
- super().testfn()

```
In [3]: class Person:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname
    def testfn(self):
        print(self.fname + self.lname)
X = Person('K', 'A')
X.testfn()

class Student(Person):
    def __init__(self, fname, lname, year):
        Person.__init__(self, fname, lname) # Person.__init__(self, fname, lname)
        self.year = year
    def testfn2(self):
        Person.testfn(self) #Person.testfn(self)
        return self.fname + self.lname + str(self.year)

S1 = Student("b", "a", 2024)
S1.testfn2()
```

KA
ba

```
Out[3]: 'ba2024'
```

```
In [4]: class Person:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname
    def testfn(self):
        print(self.fname + self.lname)
X=Person('K', 'A')
X.testfn()

class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname) #super().__init__(fname, lname)
        self.year = year
    def testfn2(self):
        super().testfn() #super().testfn()
        return self.fname + self.lname + str(self.year)

S1=Student("b", "a", 2024)
S1.testfn2()
```

```
KA
ba
'ba2024'
```

In Python, `super()` is a built-in function that allows you to call a method from the parent or sibling class. It is a way to access and call functions of a class that is inherited by another class. This function is particularly useful in cases where we have multiple inheritance or class hierarchies.

Here's an example of how to use `super()` in Python:

```
# Define a parent class
class Parent:
    def __init__(self):
        print("Parent's constructor called")

    def print_message(self):
        print("This is a message from the Parent class.")

# Define a child class that inherits from the parent class
class Child(Parent):
    def __init__(self):
        # Call the parent's constructor using super()
        super().__init__()
        print("Child's constructor called")

    # Override the print_message method from the parent class
    def print_message(self):
        # Call the parent's print_message method using super()
        super().print_message()
        print("This is an additional message from the Child class.")

# Create an object of the child class and call the print_message method
child = Child()
child.print_message()
```

In this example, `super()` is used to call the `__init__()` method of the `Parent` class from the `Child` class. It is also used to call the `print_message()` method of the `Parent` class from the `Child` class. This ensures that the methods of the parent class are still executed when the methods of the child class are called. The output of this code will be:

```
Parent's constructor called
Child's constructor called
This is a message from the Parent class.
This is an additional message from the Child class.
```

As you can see, both the parent's and child's constructors are called, and both the parent's and child's messages are printed. */

In Python, the `__init__()` method is a special method that's defined within a class. It's called when an instance (or object) of the class is created. This method is used to initialize the attributes of the class.

The `Classname.__init__(self, attribute1, attribute2)` part of your prompt is used to call the `__init__` method of the parent class (also called the superclass) from within the child class. This is done to ensure that the parent class's `__init__` method is executed when a new instance of the child class is created.

The `super().__init__(attribute1, attribute2)` part of your prompt is used to call the `__init__` method of the parent class. The `super()` function returns a proxy object that delegates method calls to the parent class.

Here's an example to illustrate the usage:

```
# Define the parent class
class Parent:
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
```

```

self.attribute2 = attribute2

def print_attributes(self):
print("Attribute 1:", self.attribute1)
print("Attribute 2:", self.attribute2)

# Define the child class
class Child(Parent): # Child inherits from Parent
def __init__(self, attribute1, attribute2, attribute3):
# Call the parent class's __init__ method
super().__init__(attribute1, attribute2)

# Initialize the attribute of the child class
self.attribute3 = attribute3

def print_attributes(self):
# Call the parent class's method
super().print_attributes()

# Print the attribute of the child class
print("Attribute 3:", self.attribute3)

# Create an instance of the child class
child_instance = Child("Attribute 1 value", "Attribute 2 value", "Attribute 3 value")

# Call the child class's method
child_instance.print_attributes()

```

This code will output:

```

Attribute 1: Attribute 1 value
Attribute 2: Attribute 2 value
Attribute 3: Attribute 3 value

```

- In this example, the `Child` class inherits from the `Parent` class. The `super().__init__(attribute1, attribute2)` line in the `Child` class's `__init__` method ensures that the `Parent` class's `__init__` method is called when a new instance of the `Child` class is created.
- Similarly, the `super().print_attributes()` line in the `Child` class's `print_attributes` method ensures that the `Parent` class's `print_attributes` method is called when the `print_attributes` method of the `Child` class instance is called.

*/

```
In [2]: class User:
    def signin(self):
        print("Loggedin")
```

IMP:

Now you might be wondering, Where is the `init` method here. Shouldn't we have that `init` method that gets run first.

Well we could, if we don't have any variables or attributes that we want to assign to the user.

Well in that case we wouldn't need an `init` method.

```
In [6]: # How can we make sure that all of these users or classes also have access to sign in.
# Well we can use inheritance.
# users:-
#     1.Archers
#     2.Wizards
#     3.Ogres
class User():
    def signin(self):
        return "Logged-in"
class Wizard(User):
    pass
class Archer(User):
    pass
wizard1=Wizard()
print(wizard1.signin())
```

Logged-in

```
In [4]: class User():
    def signin(self):
        return "Logged-in"

class Wizard(User):
    def __init__(self, name, power):
        self.name=name
        self.power=power
    def attack(self):
        return f"attacking with the power of : {self.power}")

class Archer(User):
```

```

def __init__(self, name, num_arrows):
    self.name=name
    self.num_arrows=num_arrows
def attack(self):
    return(f"attacking with the arrows,arrows left-{self.num_arrows}")

wizard1=Wizard('Merlin',50)
archer1=Archer('joe',500)
print(wizard1.signin())
print(wizard1.attack())
print(archer1.signin())
print(archer1.attack())

```

```

Logged-in
attacking with the power of : 50
Logged-in
attacking with the arrows,arrows left-500

```

isinstance()

Python gives us a useful tool to check if something is an instance of a class and easily enough for us. It's called `isinstance` and `isinstance` is a built in function in Python we give it the instance and then the class that we want to check.

Syntax: `isinstance(instance,Class)`

```

In [8]: class User():
    def signin(self):
        return("Logged-in")

class Wizard(User):
    def __init__(self, name, power):
        self.name=name
        self.power=power
    def attack(self):
        return(f"attacking with the power of : {self.power}")

class Archer(User):
    def __init__(self, name, num_arrows):
        self.name=name
        self.num_arrows=num_arrows
    def attack(self):
        return(f"attacking with the arrows,arrows left-{self.num_arrows}")

wizard1=Wizard('Merlin',50)
archer1=Archer('joe',500)
print(wizard1.signin())
print(isinstance(wizard1,Wizard))
print(isinstance(wizard1,User))

```

```

Logged-in
True
True

```

Well in Python., Remember how I said everything is an object and everything in Python inherits from the base object class that Python comes with And it's called object.

Let's click Run it's true because wizard 1 inherits or gets methods from the Wizard class from the user class and even higher up from the object based class that Python comes with.

```

In [14]: class User():
    def signin(self):
        return("Logged-in")

class Wizard(User):
    def __init__(self, name, power):
        self.name=name
        self.power=power
    def attack(self):
        return(f"attacking with the power of: {self.power}")

class Archer(User):
    def __init__(self, name, num_arrows):
        self.name=name
        self.num_arrows=num_arrows
    def attack(self):
        return(f"attacking with the arrows,arrows left-{self.num_arrows}")

wizard1=Wizard('Merlin',50)
print(isinstance(wizard1,object))

```

```
True
```

Polymorphism:

<https://www.geeksforgeeks.org/polymorphism-in-python/>

Polymorphism in python defines methods in the child class that have the same name as the methods in the parent class.

In inheritance, the child class inherits the methods from the parent class. Also, it is possible to modify a method in a child class that it has inherited from the parent class.

Polymorphism means many forms. Now we know that methods belong to objects. We use the `self` keyword to act upon the object that got instantiated.

This idea of polymorphism refers to the way in which object classes can share the same method name but those method names can act differently based on what object calls them.

Polymorphism in Python refers to the ability of a single function or method to handle different types of arguments, or for a class to be used in multiple ways. This is achieved through the use of Duck Typing, which is a principle in Python that allows you to use an object based on what methods it defines, rather than its actual class.

Here's an example of polymorphism in Python using Duck Typing:

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

print(add(1, 2)) # Output: 3
print(add('Hello, ', 'World!')) # Output: Hello, World!
```

In this example, the `add` function can take two arguments of different types (integers and strings) and still perform addition. This is polymorphism in action. The function doesn't check the types of `a` and `b` before performing addition; instead, it assumes that the objects it receives have an `add` method defined. If the objects don't support addition, a `TypeError` will be raised. This is the essence of Duck Typing.

Here's another example of polymorphism using classes:

```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return 'Woof!'

class Cat(Animal):
    def speak(self):
        return 'Meow!'

def animal_speak(animal):
    return animal.speak()

dog = Dog()
cat = Cat()

print(animal_speak(dog)) # Output: Woof!
print(animal_speak(cat)) # Output: Meow!
```

In this example, `Dog` and `Cat` classes both inherit from the `Animal` class and override the `speak` method.

The `animal_speak` function takes an `Animal` object and calls its `speak` method. Since both `Dog` and `Cat` are instances of `Animal`, we can pass them to `animal_speak` and it will call their respective `speak` methods, demonstrating polymorphism.

`animal_speak(dog)` `return` equals to `dog.speak()`

```
dog = Dog()
dog.speak() --- returns 'woof'
```

Used to visualize the code execution: <https://pythontutor.com/python-compiler.html#mode=edit>

```
In [1]: class User():
    def signin(self):
        return("Logged-in")

class Wizard(User):
    def __init__(self, name, power):
        self.name=name
        self.power=power
    def attack(self):
        return(f"attacking with the power of : {self.power}")

class Archer(User):
    def __init__(self, name, num_arrows):
        self.name=name
        self.num_arrows=num_arrows
    def attack(self):
        return(f"attacking with the arrows, arrows left-{self.num_arrows}")

wizard1=Wizard('Merlin',50)
archer1=Archer('joe',500)
```

```
print(wizard1.signin())
print(wizard1.attack())
print(archer1.signin())
print(archer1.attack())
```

```
Logged-in
attacking with the power of : 50
Logged-in
attacking with the arrows,arrows left-500
```

```
In [9]: class User():
    def signin(self):
        return("Logged-in")

class Wizard(User):
    def __init__(self,name,power):
        self.name=name
        self.power=power
    def attack(self):
        print(f"{self.name} attacking with the power of : {self.power}")

class Archer(User):
    def __init__(self,name,num_arrows):
        self.name=name
        self.num_arrows=num_arrows
    def attack(self):
        print(f"{self.name} attacking with the arrows,arrows left-{self.num_arrows}")

wizard1=Wizard('Merlin',50)
archer1=Archer('Joe',500)

def player_attack(char):
    char.attack()

player_attack(wizard1)
player_attack(archer1)
```

```
Merlin attacking with the power of : 50
Joe attacking with the arrows,arrows left-500
```

```
In [11]: class User():
    def signin(self):
        return("Logged-in")

class Wizard(User):
    def __init__(self,name,power):
        self.name=name
        self.power=power
    def attack(self):
        print(f"{self.name} attacking with the power of : {self.power}")

class Archer(User):
    def __init__(self,name,num_arrows):
        self.name=name
        self.num_arrows=num_arrows
    def attack(self):
        print(f"{self.name} attacking with the arrows,arrows left-{self.num_arrows}")

wizard1=Wizard('Merlin',50)
archer1=Archer('Joe',500)

for char in [wizard1,archer1]:
    char.attack()
```

```
Merlin attacking with the power of : 50
Joe attacking with the arrows,arrows left-500
```

let's say that the user had a attack method in this default attack method is let's say print do nothing because it's just a user even, if I run these and let's say print wizard1.attack().

If I click run it's going to override whatever the original attack was because we already have that method in our wizard class.

```
In [12]: class User():
    def signin(self):
        return("Logged-in")
    def attack(self):
        print("Do nothing")

class Wizard(User):
    def __init__(self,name,power):
        self.name=name
        self.power=power
    def attack(self):
        print(f"{self.name} attacking with the power of : {self.power}")

class Archer(User):
    def __init__(self,name,num_arrows):
        self.name=name
        self.num_arrows=num_arrows
    def attack(self):
        print(f"{self.name} attacking with the arrows,arrows left-{self.num_arrows}")

wizard1=Wizard('Merlin',50)
```

```
archer1=Archer('Joe',500)
for char in [wizard1,archer1]:
    char.attack()
```

Merlin attacking with the power of : 50
 Joe attacking with the arrows,arrows left-500

IMP:

But let's say I wanted to have both user and Wizard run the attack method.

How can we do this? for now I can say user.attack and give it self because I accept the user as my parameter in here. So, polymorphism allows us to have many forms.

```
In [13]: class User():
    def signin(self):
        return("Logged-in")
    def attack(self):
        print("Do nothing")

class Wizard(User):
    def __init__(self,name,power):
        self.name=name
        self.power=power
    def attack(self):
        User.attack(self)
        print(f"{self.name} attacking with the power of : {self.power}")

class Archer(User):
    def __init__(self,name,num_arrows):
        self.name=name
        self.num_arrows=num_arrows
    def attack(self):
        print(f"{self.name} attacking with the arrows,arrows left-{self.num_arrows}")

wizard1=Wizard('Merlin',50)
archer1=Archer('Joe',500)
print(wizard1.attack())
```

Do nothing
 Merlin attacking with the power of : 50
 None

Polymorphism:

It is the ability to redefine methods for these derived classes that is wizard and Archer and an object that gets instantiated can behave in different forms in different ways based on polymorphism.

Exercise:Pets Everywhere

```
In [1]: class Pets():
    animals = []
    def __init__(self, animals):
        self.animals = animals

    def walk(self):
        for animal in self.animals:
            print(animal.walk())

class Cat():
    is_lazy = True
    def __init__(self, name, age):
        self.name = name
        self.age = age

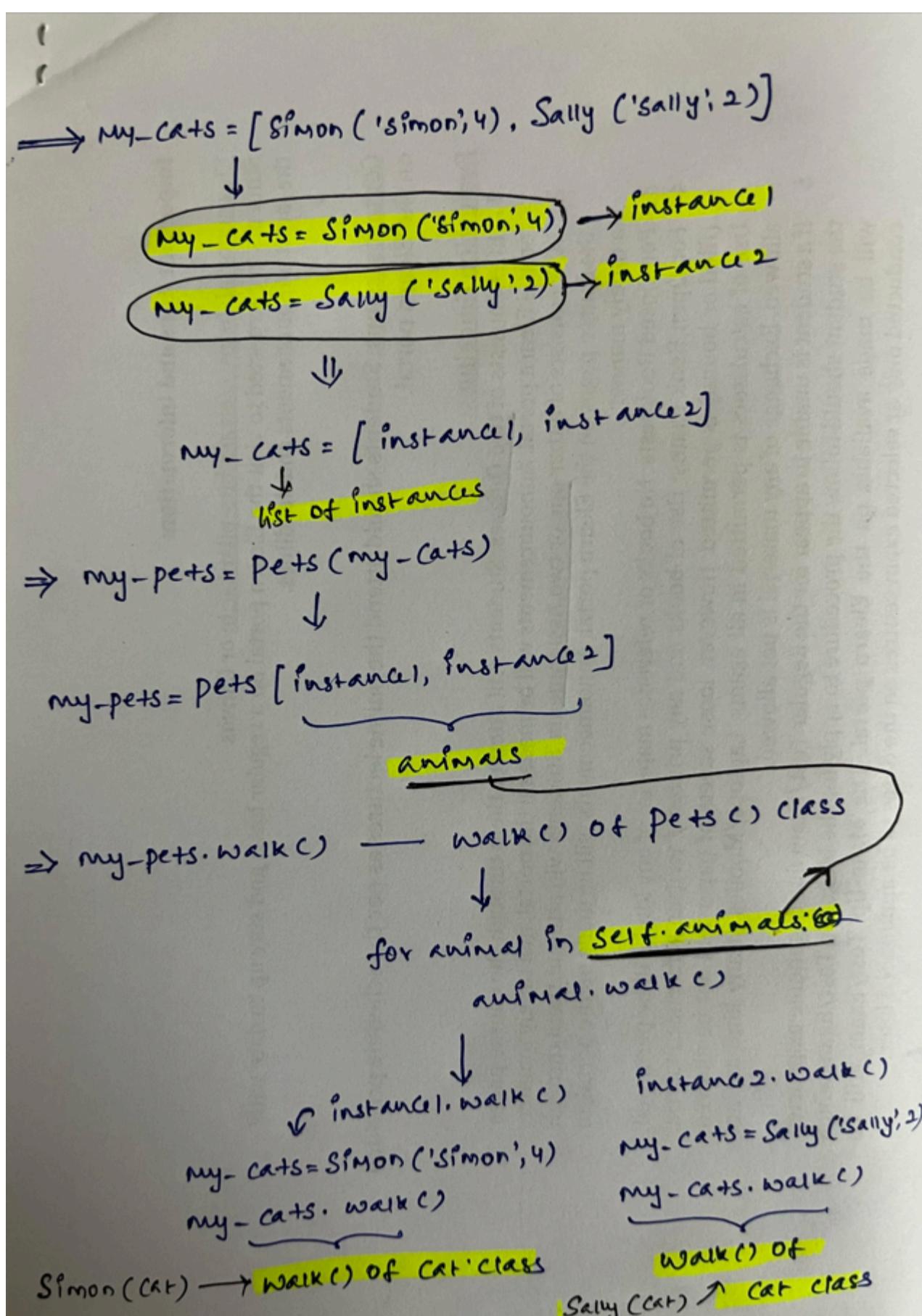
    def walk(self):
        return f'{self.name} is just walking around'

class Simon(Cat):
    def sing(self, sounds):
        return f'{sounds}'
class Sally(Cat):
    def sing(self, sounds):
        return f'{sounds}'

#1 Add another Cat
class Suzy(Cat):
    def sing(self, sounds):
        return f'{sounds}'

#2 Create a list of all of the pets (create 3 cat instances from the above)
my_cats = [Simon('Simon', 4), Sally('Sally', 21), Suzy('Suzy', 1)]
#3 Instantiate the Pet class with all your cats
my_pets = Pets(my_cats)
#4 Output all of the cats singing using the my_pets instance
my_pets.walk()
```

Simon is just walking around
 Sally is just walking around
 Suzy is just walking around



```

my_cats = [Simon('Simon', 4), Sally('Sally', 21), Suzy('Suzy', 1)]
my_pets = Pets(my_cats)
my_pets.walk()
    
```

Certainly! Let's delve into a more detailed explanation of how each line of code is executed:

1. Creating Instances of Cat Subclasses:

- Three instances of the `Cat` subclasses (`Simon`, `Sally`, and `Suzy`) are created with specific names and ages:
- ```

Simon('Simon', 4)
Sally('Sally', 21)
Suzy('Suzy', 1)

```
- When each instance is created, the `__init__` method of the respective subclass (`Simon`, `Sally`, or `Suzy`) is called, initializing the instance with the provided name and age.

### 2. Initializing the `my_cats` List:

- These instances are then stored in a list called `my_cats`:
- ```

my_cats = [Simon('Simon', 4), Sally('Sally', 21), Suzy('Suzy', 1)]
    
```
- The list `my_cats` now contains the three instances of cat subclasses, `Simon`, `Sally`, and `Suzy`.

3. Instantiating `Pets` Class:

- The `Pets` class is instantiated with the list of cat instances (`my_cats`):
- ```

my_pets = Pets(my_cats)

```
- This calls the `__init__` method of the `Pets` class, passing `my_cats` as an argument. Inside the `__init__` method, `self.animals` is set to `my_cats`, effectively storing the list of cat instances within the `my_pets` instance.

### 4. Calling `walk` Method:

- The `walk` method of the `Pets` class is called on the `my_pets` instance:  
`my_pets.walk()`
- This triggers the execution of the `walk` method defined within the `Pets` class.
- Inside the `walk` method:
  - A loop iterates over each element (`animal`) in the `self.animals` list (which contains instances of cat subclasses).
  - For each `animal`, the `walk` method of that specific cat instance is called (`animal.walk()`), and the result is printed out.

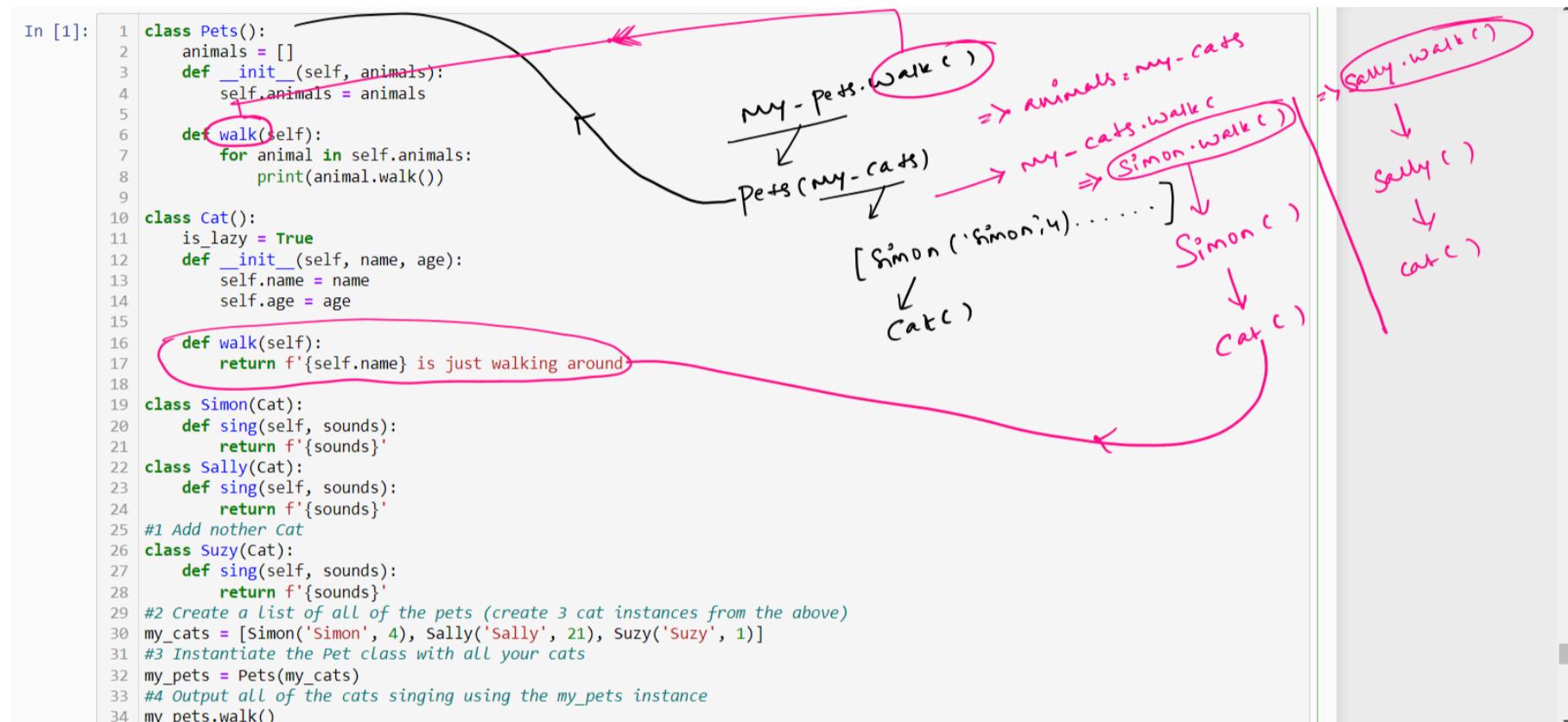
## 5. Execution of Cat's `walk` Method:

- For each cat instance in `my_cats`, the `walk` method defined in the `Cat` class (or its subclasses) is executed.
- The `walk` method simply returns a string indicating that the cat is walking around, including the cat's name.
- This string is printed out within the `walk` method of the `Pets` class.

## 6. Final Output:

- The output would be printed lines indicating that each cat is just walking around, with their respective names:
- Simon is just walking around  
 Sally is just walking around  
 Suzy is just walking around

That's how each line of code is executed in detail, from creating instances of cat subclasses to printing out the walking messages of each cat.



```
In [13]: class Pets():
 animals = []
 def __init__(self, animals):
 self.animals = animals

 def walk(self):
 for animal in self.animals:
 print(animal.walk())

class Cat():
 is_lazy = True

 def __init__(self, name, age):
 self.name = name
 self.age = age

 def walk(self):
 return f'{self.name} is just walking around'

#2 Create a list of all of the pets (create 3 cat instances from the above)
my_cats = [Cat('Simon', 4), Cat('Sally', 21), Cat('Suzy', 1)]

#3 Instantiate the Pet class with all your cats
my_pets = Pets(my_cats)

#4 Output all of the cats singing using the my_pets instance
my_pets.walk()
```

Simon is just walking around  
 Sally is just walking around  
 Suzy is just walking around

```

1 class Pets():
2 animals = []
3 def __init__(self, animals):
4 self.animals = animals
5
6 def walk(self):
7 for animal in self.animals:
8 print(animal.walk())
9
10 class Cat():
11 is_lazy = True
12
13 def __init__(self, name, age):
14 self.name = name
15 self.age = age
16
17 def walk(self):
18 return f'{self.name} is just walking around'
19
20 #2 Create a list of all of the pets (create 3 cat instances from the above)
21 my_cats = [Cat('Simon', 4), Cat('Sally', 21), Cat('Suzy', 1)]
22
23 #3 Instantiate the Pet class with all your cats
24 my_pets = Pets(my_cats)
25
26 #4 Output all of the cats singing using the my_pets instance
27 my_pets.walk()

```

Simon is just walking around  
Sally is just walking around  
Suzy is just walking around

## Note:

The link is the animals list. Each object in that list is from a class that was derived from the Cat class, so they inherited the .walk() method from Cat.

The Pets .walk() method goes through the list and calls the .walk() method for each object.

## Super()

```
In [4]: class User():
 def __init__(self, email):
 self.email=email
 def signin(self):
 return("Logged-in")

class Wizard(User):
 def __init__(self,name,power,email):
 User.__init__(self,email)
 self.name=name
 self.power=power
 def attack(self):
 return(f"{self.name} attacking with the power of : {self.power}")

wizard1=Wizard('Merlin',50,"check@gmail.com")

print(wizard1.signin())
print(wizard1.attack())
print(wizard1.email)
```

Logged-in  
Merlin attacking with the power of : 50  
check@gmail.com

```
In [6]: class User():
 def __init__(self, email):
 self.email=email
 def signin(self):
 return("Logged-in")

class Wizard(User):
 def __init__(self,name,power,email):
 super().__init__(email)
 self.name=name
 self.power=power
 def attack(self):
 return(f"{self.name} attacking with the power of : {self.power}")

wizard1=Wizard('Merlin',50,"check@gmail.com")

print(wizard1.signin())
print(wizard1.attack())
print(wizard1.email)
```

Logged-in  
Merlin attacking with the power of : 50  
check@gmail.com

## Object Introspection:

Introspection in computer programming means the ability to determine the type of an object at runtime.

What is runtime - That is when the code is running.

You can determine the type of an object and it's actually one of Python's strengths because everything in Python is an object.

Object introspection in Python refers to the ability of an object to reflect upon and understand its own structure and properties. This capability is supported by the built-in Python modules like `inspect`, `dir()`, `hasattr()`, `getattr()`, and `setattr()`. These modules allow the programmer to understand the contents of an object, such as its methods, properties, and values.

For example, the `dir()` function can be used to get a list of attributes and methods of an object:

```
class Example:
 def method(self):
 pass
```

```
obj = Example()
print(dir(obj))
```

This will output: `['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'method']`

The `getattr()` function can be used to access an attribute of an object, and `setattr()` function to change or add an attribute:

```
class Example:
 def __init__(self, value):
 self.value = value

obj = Example(5)
print(getattr(obj, 'value')) # Output: 5

setattr(obj, 'value', 10)
print(getattr(obj, 'value')) # Output: 10
```

The `hasattr()` function can be used to check if an object has a specific attribute:

```
class Example:
 def __init__(self, value):
 self.value = value

obj = Example(5)
print(hasattr(obj, 'value')) # Output: True
print(hasattr(obj, 'nonexistent_value')) # Output: False
```

These features are particularly useful in metaprogramming, where the code itself generates other code, allowing for greater flexibility and dynamism.

Python allows us to do introspection and inspect these objects with some nice helper functions this one function is called `dir`.

if I run this will give me all of the methods and attributes that the wizard instant has.

```
In [11]: class User():
 def __init__(self, email):
 self.email=email
 def signin(self):
 return("Logged-in")

class Wizard(User):
 def __init__(self,name,power,email):
 super().__init__(email)
 self.name=name
 self.power=power
 def attack(self):
 return(f"{self.name} attacking with the power of : {self.power}")

wizard1=Wizard('Merlin',50,"check@gmail.com")

print(dir(wizard1)) #introspection
wizard1.email
```

[`'__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',  
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',  
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'attack', 'email', 'name', 'power',  
'signin']  
Out[11]: 'check@gmail.com'`

## Dunder Methods:

<https://docs.python.org/3/reference/datamodel.html#special-method-names>

These double underscore Dunder methods are special methods that Python recognizes.

```
In [8]: class Toy():
 def __init__(self, color, age):
 self.color = color
 self.age = age
 def signin(self):
 return "Logged-in"
action = Toy("black", 2)
print(dir(action))

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'age', 'color', 'signin']
```

```
In [11]: class Toy():
 def __init__(self, color, age):
 self.color = color
 self.age = age
action = Toy("black", 2)
print(action.__str__)
print(action.__str__())
print(str(action))

<method-wrapper '__str__' of Toy object at 0x00000160B4C51760>
<__main__.Toy object at 0x00000160B4C51760>
<__main__.Toy object at 0x00000160B4C51760>
```

```
In [1]: class Toy():
 def __init__(self, color, age):
 self.color = color
 self.age = age
 def __str__(self): # changing the __str__ method
 return f'{self.color}'

action = Toy("black", 2)
print(action.__str__)
print(action.__str__())
print(str(action))

<bound method Toy.__str__ of <__main__.Toy object at 0x000001C2FCF4FBB0>>
black
black
```

```
In [1]: class Toy():
 def __init__(self, color, age):
 self.color = color
 self.age = age
 def __str__(self): # changing the __str__ method
 return f'{self.color}'
 def __len__(self):
 return 5
 def __del__(self):
 print("deleted!!!")
 def __call__(self):
 print("yes??")
action = Toy("black", 2)
print(action.__str__())
print(str(action))
print(len(action))
del action # delete keyword deletes some sort of variable that we might have had in our program.
print(action())
```

```
black
black
5
deleted!!!

NameError Traceback (most recent call last)
<ipython-input-1-a9d83324bcd0> in <module>
 16 print(len(action))
 17 del action # delete keyword deletes some sort of variable that we might have had in our program.
--> 18 print(action())

NameError: name 'action' is not defined
```

I have this special way to call a function which by the way underneath the hood the way we're able to call functions is using this Dunder call

```
In [12]: class Toy():
 def __init__(self, color, age):
 self.color = color
 self.age = age
 self.my_dict = {'name': 'Yoyo', 'has_pets': False}
 def __str__(self): # changing the __str__ method
 return f'{self.color}'
 def __len__(self):
 return 5
 def __del__(self):
 print("deleted!!!")
 def __call__(self):
 print("yes??")
```

```

def __getitem__(self,i):
 return self.my_dict[i]
action=Toy("black",2)
print(action.__str__())
print(str(action))
print(len(action))
del action #delete keyword deletes some sort of variable that we might have had in our program.
print(action())
print(action['name'])

deleted!!!
black
black
5
yes??
None
Yoyo

```

### Exercise: Extending List:

```
In [16]: class SuperList():
 def __len__(self):
 return 1000

super_list1 = SuperList();

print(len(super_list1))

1000
```

```
In [17]: class SuperList():
 def __len__(self):
 return 1000

super_list1 = SuperList();

print(len(super_list1))
super_list1.append(5)
print(super_list1[0])

1000
```

```

AttributeError Traceback (most recent call last)
<ipython-input-17-7232bb1d9ccf> in <module>
 6
 7 print(len(super_list1))
----> 8 super_list1.append(5)
 9 print(super_list1[0])

AttributeError: 'SuperList' object has no attribute 'append'
```

Python is an object that inherits from the base object class we then inherit some built in list methods

```
In [20]: class SuperList(list):
 def __len__(self):
 return 1000

super_list1 = SuperList();

print(len(super_list1))
super_list1.append(5)
print(super_list1[0])
print(issubclass(SuperList,list))
print(issubclass(list, object))

1000
5
True
True
```

Dunder methods, short for "double underscore methods", are special methods in Python that are predefined and automatically available for all objects. They are also known as "magic methods" or "special methods" because they allow you to customize the behavior of the built-in operations in Python.

Dunder methods are named with a double underscore prefix and suffix, such as `__init__` or `__add__`. Here's an example of how they can be used:

1. `__init__`: This method is called when an object is created from a class and it allows the class to initialize the attributes of the class.

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

p = Person("John", 30)
print(p.name) # Output: John
print(p.age) # Output: 30
```

2. `__str__`: This method is called when we try to convert an object into a string.

```

class Person:
 def __init__(self, name, age):
 self.name = name
 self.age = age

 def __str__(self):
 return f"{self.name} is {self.age} years old"

p = Person("John", 30)
print(str(p)) # Output: John is 30 years old

```

3. `__add__`: This method is called when the addition operator (+) is used on objects of the class.

```

class Vector:
 def __init__(self, x, y):
 self.x = x
 self.y = y

 def __add__(self, other):
 return Vector(self.x + other.x, self.y + other.y)

v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2
print(v3.x, v3.y) # Output: 4 6

```

These are just a few examples of dunder methods. Python has many more built-in dunder methods that you can override to customize the behavior of your classes.

## Types Of Inheritance:

In Python, there are four types of inheritance: Single, Multiple, Multilevel, Hierarchical and Hybrid. Let's discuss each one with examples.

1. Single Inheritance:

In single inheritance, a class inherits only from one parent class. For example, if there are classes `Vehicle` and `Car`, `Car` can inherit properties and methods from `Vehicle`.

```

class Vehicle:
 def general_usage(self):
 print("Most vehicles are used for transportation")

class Car(Vehicle):
 def specific_usage(self):
 print("Specific use: Commute to work, vacation with family")

car = Car()
car.general_usage() # Output: Most vehicles are used for transportation
car.specific_usage() # Output: Specific use: Commute to work, vacation with family

```

2. Multiple Inheritance:

In multiple inheritance, a class can inherit from more than one parent class. For example, classes `Engine`, `Wheels`, and `Car` can inherit from all these parent classes.

```

class Engine:
 def start_engine(self):
 print("Engine started")

class Wheels:
 def roll(self):
 print("Rolling on wheels")

class Car(Engine, Wheels):
 def drive(self):
 print("Car is driving")

car = Car()
car.start_engine() # Output: Engine started
car.roll() # Output: Rolling on wheels
car.drive() # Output: Car is driving

```

3. Multilevel Inheritance:

In multilevel inheritance, features of grandparent classes are inherited into the child class through the parent class. For example, classes `Vehicle`, `Car` (inherits from `Vehicle`), and `ElectricCar` (inherits from `Car`) form a multilevel inheritance.

```

class Vehicle:
 def general_usage(self):
 print("Most vehicles are used for transportation")

class Car(Vehicle):

```

```

def specific_usage(self):
 print("Specific use: Commute to work, vacation with family")

class ElectricCar(Car):
 def electric_power(self):
 print("Powered by electricity")

electric_car = ElectricCar()
electric_car.general_usage() # Output: Most vehicles are used for transportation
electric_car.specific_usage() # Output: Specific use: Commute to work, vacation with family
electric_car.electric_power() # Output: Powered by electricity

```

#### 4. Hierarchical Inheritance:

In hierarchical inheritance, there is a parent class and multiple child classes. For example, classes `Vehicle` (parent) and `Car`, `Bike` (children) inherit from the `Vehicle` class.

```

class Vehicle:
 def general_usage(self):
 print("Most vehicles are used for transportation")

class Car(Vehicle):
 def specific_usage(self):
 print("Specific use: Commute to work, vacation with family")

class Bike(Vehicle):
 def specific_usage(self):
 print("Specific use: Exercise, transportation to work")

car = Car()
car.general_usage() # Output: Most vehicles are used for transportation
car.specific_usage() # Output: Specific use: Commute to work, vacation with family

bike = Bike()
bike.general_usage() # Output: Most vehicles are used for transportation
bike.specific_usage() # Output: Specific use: Exercise, transportation to work

```

#### 5. Hybrid Inheritance:

Hybrid inheritance is a combination of two or more types of inheritance. For example, in the following code, `Vehicle` is the parent class for `Car` and `Bike`, and `Car` is the parent class for `ElectricCar`. This forms a combination of hierarchical and multilevel inheritance.

```

class Vehicle:
 def general_usage(self):
 print("Most vehicles are used for transportation")

class Car(Vehicle):
 def specific_usage(self):
 print("Specific use: Commute to work, vacation with family")

class ElectricCar(Car):
 def electric_power(self):
 print("Powered by electricity")

class Bike(Vehicle):
 def specific_usage(self):
 print("Specific use: Exercise, transportation to work")

car = ElectricCar()
car.general_usage() # Output: Most vehicles are used for transportation
car.specific_usage() # Output: Specific use: Commute to work, vacation with family
car.electric_power() # Output: Powered by electricity

bike = Bike()
bike.general_usage() # Output: Most vehicles are used for transportation
bike.specific_usage() # Output: Specific use: Exercise, transportation to work

```

In Python, inheritance is a powerful feature that allows a class to inherit properties and behavior from another class. There are different types of inheritance relationships that can exist between classes. Let's discuss them with examples, including variables and methods.

## 1. Single Inheritance:

In single inheritance, a subclass inherits from only one superclass.

```

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

 def speak(self):
 raise NotImplementedError("Subclass must implement abstract method")

```

```

class Dog(Animal):
 def speak(self):
 return f"{self.name} says Woof!"

Example Usage
dog = Dog("Buddy")
print(dog.speak()) # Output: Buddy says Woof!

```

## 2. Multiple Inheritance:

In multiple inheritance, a subclass inherits from more than one superclass.

```

class Flyable:
 def fly(self):
 return "I can fly!"

class Bird:
 def __init__(self, name):
 self.name = name

 def speak(self):
 return f"{self.name} says Chirp!"

class FlyingBird(Bird, Flyable):
 pass

Example Usage
bird = FlyingBird("Sparrow")
print(bird.speak()) # Output: Sparrow says Chirp!
print(bird.fly()) # Output: I can fly!

```

## 3. Multilevel Inheritance:

In multilevel inheritance, a subclass inherits from a superclass, and another subclass inherits from this subclass, forming a hierarchy.

```

class Animal:
 def breathe(self):
 return "I can breathe!"

class Mammal(Animal):
 def walk(self):
 return "I can walk!"

class Dog(Mammal):
 def speak(self):
 return "Woof!"

Example Usage
dog = Dog()
print(dog.breathe()) # Output: I can breathe!
print(dog.walk()) # Output: I can walk!
print(dog.speak()) # Output: Woof!

```

## 4. Hierarchical Inheritance:

In hierarchical inheritance, multiple subclasses inherit from the same superclass.

```

class Animal:
 def breathe(self):
 return "I can breathe!"

class Dog(Animal):
 def speak(self):
 return "Woof!"

class Cat(Animal):
 def speak(self):
 return "Meow!"

Example Usage
dog = Dog()
cat = Cat()
print(dog.breathe()) # Output: I can breathe!
print(dog.speak()) # Output: Woof!
print(cat.breathe()) # Output: I can breathe!
print(cat.speak()) # Output: Meow!

```

These examples demonstrate the different types of inheritance in Python along with variables and methods included in each. Inheritance provides a way to create more complex and specialized classes by building upon existing ones, promoting code reuse and maintainability.

Let's modify the examples to include variables along with methods in each type of inheritance.

## 1. Single Inheritance:

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

 def speak(self):
 raise NotImplementedError("Subclass must implement abstract method")

class Dog(Animal):
 def __init__(self, name, breed):
 super().__init__(name)
 self.breed = breed

 def speak(self):
 return f"{self.name} the {self.breed} says Woof!"

Example Usage
dog = Dog("Buddy", "Labrador")
print(dog.speak()) # Output: Buddy the Labrador says Woof!
print(dog.species) # Output: Animal
```

In this example, the `Animal` class has an additional attribute `species`, while the `Dog` class adds the attribute `breed`. Instances of `Dog` inherit both attributes from the `Animal` class.

## 2. Multiple Inheritance:

```
class Flyable:
 def __init__(self, altitude):
 self.altitude = altitude

 def fly(self):
 return f"I can fly at {self.altitude} feet!"

class Bird:
 def __init__(self, name):
 self.name = name
 self.species = "Bird"

 def speak(self):
 return f"{self.name} says Chirp!"

class FlyingBird(Bird, Flyable):
 def __init__(self, name, altitude):
 Bird.__init__(self, name)
 Flyable.__init__(self, altitude)

Example Usage
bird = FlyingBird("Sparrow", 10000)
print(bird.speak()) # Output: Sparrow says Chirp!
print(bird.fly()) # Output: I can fly at 10000 feet!
print(bird.species) # Output: Bird
print(bird.altitude) # Output: 10000
```

In this example, the `FlyingBird` class inherits attributes from both `Bird` and `Flyable` classes.

## 3. Multilevel Inheritance:

```
class Animal:
 def __init__(self):
 self.species = "Animal"

class Mammal(Animal):
 def __init__(self):
 super().__init__()
 self.category = "Mammal"

class Dog(Mammal):
 def __init__(self, name):
 super().__init__()
 self.name = name

 def speak(self):
 return f"{self.name} says Woof!"

Example Usage
dog = Dog("Buddy")
print(dog.species) # Output: Animal
```

```
print(dog.category) # Output: Mammal
print(dog.speak()) # Output: Buddy says Woof!
```

In this example, the `Dog` class inherits `species` from `Animal` and `category` from `Mammal`.

## 4. Hierarchical Inheritance:

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

class Dog(Animal):
 def __init__(self, name):
 super().__init__("Canine")
 self.name = name

 def speak(self):
 return f"{self.name} says Woof!"

class Cat(Animal):
 def __init__(self, name):
 super().__init__("Feline")
 self.name = name

 def speak(self):
 return f"{self.name} says Meow!"

Example Usage
dog = Dog("Buddy")
print(dog.species) # Output: Canine
print(dog.speak()) # Output: Buddy says Woof!

cat = Cat("Whiskers")
print(cat.species) # Output: Feline
print(cat.speak()) # Output: Whiskers says Meow!
```

In this example, both `Dog` and `Cat` classes inherit the `species` attribute from the `Animal` class. Each subclass provides its own implementation of the `speak` method.

These examples demonstrate inheritance in Python with variables included along with methods, showcasing how attributes can be inherited and overridden across different inheritance types.

### Hierarchical Inheritance:

```
In [6]: class Animal:
 def __init__(self, species):
 self.species = species

class Dog(Animal):
 def __init__(self, species, name):
 super().__init__(species)
 self.name = name

 def speak(self):
 return f"{self.name} says Woof!"

class Cat(Animal):
 def __init__(self, species, name):
 super().__init__(species)
 self.name = name

 def speak(self):
 return f"{self.name} says Meow!"

Example Usage
dog = Dog('Golden', "Buddy")
print(dog.species) # Output: Canine
print(dog.speak()) # Output: Buddy says Woof!

cat = Cat('Pug', "Whiskers")
print(cat.species) # Output: Feline
print(cat.speak()) # Output: Whiskers says Meow!
```

```
Golden
Buddy says Woof!
Pug
Whiskers says Meow!
```

```
In [3]: class Animal:
 def __init__(self, species):
 self.species = species

class Dog(Animal):
 def __init__(self, name):
 super().__init__("Canine")
```

```

 self.name = name

 def speak(self):
 return f"{self.name} says Woof!"

class Cat(Animal):
 def __init__(self, name):
 super().__init__("Feline")
 self.name = name

 def speak(self):
 return f"{self.name} says Meow!"

Example Usage
dog = Dog("Buddy")
print(dog.species) # Output: Canine
print(dog.speak()) # Output: Buddy says Woof!

cat = Cat("Whiskers")
print(cat.species) # Output: Feline
print(cat.speak()) # Output: Whiskers says Meow!

```

Canine  
 Buddy says Woof!  
 Feline  
 Whiskers says Meow!

---

## Inheritance:

```

In [22]: class User():
 def signin(self):
 return("Logged-in")
 def attack(self):
 print("Do nothing")

class Wizard(User):
 def __init__(self, name, power):
 self.name=name
 self.power=power
 def attack(self):
 User.attack(self)
 print(f"{self.name} attacking with the power of : {self.power}")

class Archer(User):
 def __init__(self, name, num_arrows):
 self.name=name
 self.num_arrows=num_arrows
 def checkarrows(self):
 print(f"{self.name} attacking with the arrows, arrows left-{self.num_arrows}")
 def run(self):
 print("ran fast")

class NewClass(Wizard, Archer):
 pass

newcls1=NewClass()
newcls1.run()

```

```

TypeError Traceback (most recent call last)
<ipython-input-22-30f8912196e3> in <module>
 26 pass
 27
---> 28 newcls1=NewClass()
 29 newcls1.run()

TypeError: __init__() missing 2 required positional arguments: 'name' and 'power'

```

```

In [25]: class User():
 def signin(self):
 return("Logged-in")
 def attack(self):
 print("Do nothing")

class Wizard(User):
 def __init__(self, name, power):
 self.name=name
 self.power=power
 def attack(self):
 User.attack(self)
 print(f"{self.name} attacking with the power of : {self.power}")

class Archer(User):
 def __init__(self, name, num_arrows):
 self.name=name
 self.num_arrows=num_arrows
 def checkarrows(self):
 print(f"{self.name} attacking with the arrows, arrows left-{self.num_arrows}")
 def run(self):

```

```

 print("ran fast")

class NewClass(Wizard, Archer):
 pass

newcls1=NewClass("jack",55)
newcls1.run()
newcls1.checkarrows()

ran fast

AttributeError Traceback (most recent call last)
<ipython-input-25-aa52cb76e1ee> in <module>
 28 newcls1=NewClass("jack",55)
 29 newcls1.run()
---> 30 newcls1.checkarrows()

AttributeError: 'NewClass' object has no attribute 'checkarrows'
```

In [26]:

```

class User():
 def signin(self):
 return("Logged-in")
 def attack(self):
 print("Do nothing")

class Wizard(User):
 def __init__(self,name,power):
 self.name=name
 self.power=power
 def attack(self):
 User.attack(self)
 print(f"{self.name} attacking with the power of : {self.power}")

class Archer(User):
 def __init__(self,name,num_arrows):
 self.name=name
 self.num_arrows=num_arrows
 def checkarrows(self):
 print(f"{self.name} attacking with the arrows,arrows left-{self.num_arrows}")
 def run(self):
 print("ran fast")

class NewClass(Wizard, Archer):
 pass

newcls1=NewClass("jack",55,100)
newcls1.run()
newcls1.checkarrows()

TypeError Traceback (most recent call last)
<ipython-input-26-a65f6da73e72> in <module>
 26 pass
 27
---> 28 newcls1=NewClass("jack",55,100)
 29 newcls1.run()
 30 newcls1.checkarrows()

TypeError: __init__() takes 3 positional arguments but 4 were given
```

In [38]:

```

class User():
 def signin(self):
 return("Logged-in")

class Wizard(User):
 def __init__(self,name,power):
 self.name=name
 self.power=power
 def attack(self):
 print(f"{self.name} attacking with the power of : {self.power}")

class Archer(User):
 def __init__(self,name,arrows):
 self.name=name
 self.arrows=arrows
 def checkarrows(self):
 print(f"{self.name} attacking with the arrows,arrows left:-{self.arrows}")
 def run(self):
 print("ran fast")

class NewClass(Wizard, Archer):
 def __init__(self,name,power,arrows):
 Archer.__init__(self,name,arrows)
 Wizard.__init__(self,name,power)

newcls1=NewClass("jack",55,200)
newcls1.run()
newcls1.checkarrows()
newcls1.attack()
```

```
ran fast
jack attacking with the arrows,arrows left:-200
jack attacking with the power of : 55
```

## Method Resolution Order(MRO):

<http://www.srikanthtechnologies.com/blog/python/mro.aspx>

```
In [3]: class A:
 num = 10

class B(A):
 pass

class C(A):
 num = 1

class D(B, C):
 pass
```

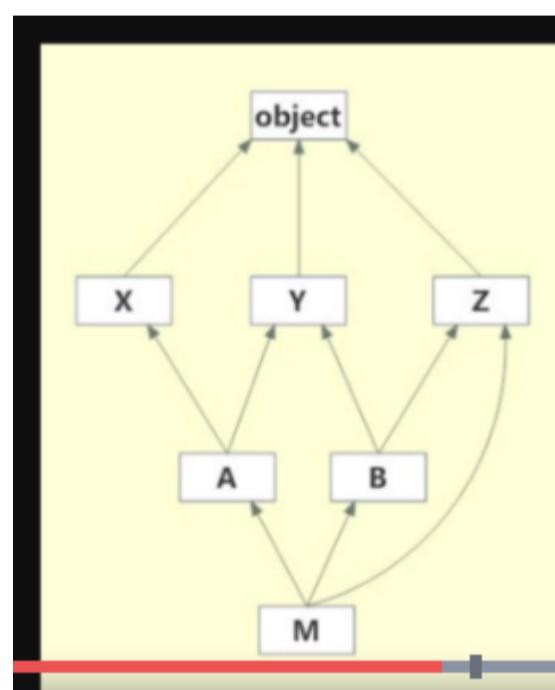


You see that D has multiple inheritance from B and C and B and C inherit from A

```
In [4]: D.mro() #gives the order of the inheritance
Out[4]: [<__main__.D, __main__.B, __main__.C, __main__.A, object>]
```

```
In [5]: print(D.__str__)
<slot wrapper '__str__' of 'object' objects>
```

```
In [6]: class X:
 pass
class Y:
 pass
class Z:
 pass
class A(X,Y):
 pass
class B(Y,Z):
 pass
class M(B,A,Z):
 pass
```



This is because of the algorithm that they use for doing MRO which is called depth first search.

```
In [7]: M.mro()
Out[7]: [<__main__.M,
 __main__.B,
 __main__.A,
 __main__.X,
 __main__.Y,
 __main__.Z,
 object>]
```

Method Resolution Order (MRO) in Python is a linear sequence of base classes that would be searched when looking for a method in a class. It's used for resolving the ambiguity that arises when a class inherits from more than one class.

Python follows the "depth-first left-to-right" rule for MRO, which means it will search depthwise left to right in the inheritance tree.

Let's take an example to understand this concept:

```
class A:
 def method(self):
 print("Inside method of A")

class B:
 def method(self):
 print("Inside method of B")

class C(A, B):
 pass

c = C()
c.method()
```

In the above code, the `C` class inherits from both `A` and `B`. If we call `c.method()`, it will create confusion as to which method to call, the one from `A` or the one from `B`. In such cases, Python follows the MRO. So, in this case, Python will first look for the method in `C`, then in `A`, and then in `B`. If the method is not found in any of these classes, it will throw a `AttributeError`.

So, the output of the above code will be:

```
Inside method of A
```

This is because Python follows the depth-first left-to-right rule, so it first checks `C`, then `A`, and then `B`. Since the method is defined in `A`, that's the one that gets called.

If we want to see the MRO of a class, we can use the `mro()` function:

```
print(C.mro())
```

The output of this will be:

```
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```

This shows the order in which Python will search for methods when we create an object of class `C`.

## Abstract Method:

An abstract method in Python is a method defined in a superclass that has no implementation in the superclass, but is intended to be implemented in subclasses. Abstract methods serve as a way to define a method signature that must be implemented by any subclass, ensuring that certain behavior is present across all subclasses.

In Python, abstract methods are created using the `@abstractmethod` decorator from the `abc` module (Abstract Base Classes). This module provides the infrastructure for defining abstract base classes and abstract methods.

Here's an example of an abstract method:

```
from abc import ABC, abstractmethod

class Animal(ABC):
 @abstractmethod
 def speak(self):
 pass

class Dog(Animal):
 def speak(self):
 return "Woof!"

class Cat(Animal):
 def speak(self):
 return "Meow!"

Example Usage
dog = Dog()
print(dog.speak()) # Output: Woof!

cat = Cat()
print(cat.speak()) # Output: Meow!
```

In this example, the `Animal` class has an abstract method `speak()`. It's defined using the `@abstractmethod` decorator, indicating that any subclass of `Animal` must provide an implementation for `speak()`. The `Dog` and `Cat` classes both implement the `speak()` method, fulfilling the requirement imposed by the abstract method in the `Animal` class.

If a subclass fails to implement an abstract method defined in its superclass, attempting to instantiate an object of that subclass will raise a `TypeError`. Abstract methods are particularly useful when you want to define a common interface for a group of related classes while leaving the specific implementation details to the individual subclasses.

Abstract methods in Python are methods that have been declared in a base class, but have not been implemented in the base class. They must be implemented in any derived class. Abstract methods are declared using the `@abstractmethod` decorator in Python.

The `abc` module in Python provides the necessary infrastructure for defining abstract base classes. Here's an example:

```
from abc import ABC, abstractmethod

class Animal(ABC):
 @abstractmethod
 def sound(self):
 pass

class Dog(Animal):
 def sound(self):
 return "Woof!"

class Cat(Animal):
 def sound(self):
 return "Meow!"

class Cow(Animal):
 def sound(self):
 return "Moo!"
```

In this example, `Animal` is an abstract base class and `Dog`, `Cat`, and `Cow` are derived classes. The `Animal` class has an abstract `sound` method, which is implemented in all derived classes.

If we try to create an object of the `Animal` class, it will result in an error:

```
a = Animal() # Error: Can't instantiate abstract class Animal with abstract method sound
```

However, we can create objects of the derived classes:

```
d = Dog()
print(d.sound()) # Output: Woof!

c = Cat()
print(c.sound()) # Output: Meow!

co = Cow()
print(co.sound()) # Output: Moo!
```

## Utils:

In Python, "utils" is a common abbreviation for "utilities" or "utility functions". Utility functions are typically small, often reusable functions that perform common tasks or provide convenient functionality that may not be readily available in the standard libraries or built-in functions.

These utility functions are often grouped together in modules or packages named "utils" or something similar. They can cover a wide range of tasks, from string manipulation to file handling, data processing, and more.

For example, a `string_utils.py` module might contain utility functions for working with strings, such as functions to capitalize words, remove whitespace, or format text in a particular way.

Here's a simple example of a utility function that reverses a string:

```
def reverse_string(s):
 """Reverse a string."""
 return s[::-1]
```

Utility functions can be quite handy as they encapsulate common operations, making your code more modular, readable, and easier to maintain. They can also be shared across different projects or modules within a project, reducing duplication of code.

In [ ]: