Cool Python Cheat Codes / Tips - 2024

A. Working Numbers

-- Assign Multiple Variables on One Line

In [9]:
```
a, b = 12, 30

# Rather
# a = 12
# b = 30

print("Value of 'a' is" , a ,"and Value of 'a' is" , b)
```

```
Value of 'a' is 12 and Value of 'a' is 30
```

-- Raise a number to a Power

In [11]:
```
power = pow(3, 5)
print("Power of 3 raise to 5: ", power)
```

```
Power of 3 raise to 5:  243
```

-- Banker's Rounding - Half towards Even

In [10]:
```
print("Round of 10.5 is" , round(10.5)) # round down to 10
print("Round of 11.5 is" , round(11.5)) # round up to 12
```

```
Round of 10.5 is 10
Round of 11.5 is 12
```

-- Using "Underscores" in Large Numbers

In [12]:
```
billion = 1_000_000_000
print("Billion :", billion)
```

```
Billion : 1000000000
```

-- Primitive data types are "Immutable"

In [10]:
```
primitive_data1 = 10
primitive_data2 = primitive_data1
primitive_data2 = 1 # replaced data2

print("Address of data1 is: ", id(primitive_data1)) # different addresses
print("Address of data1 is: ", id(primitive_data2)) # different addresses
print("Data 1 now is: ", primitive_data1)
```

```
Address of data1 is:  2615765789264
Address of data1 is:  2615765788976
Data 1 now is:  10
```

-- Check if exists

Check if exists

In [16]:
```
# Set the variable 'age' to the value 16
age = 16
print("Initial value of 'age':", age)

# Check if 'age' is in the local and global namespaces
print("'age' in locals():", 'age' in locals())
print("'age' in globals():", 'age' in globals())

# Delete the variable 'age'
del age
# print("After deleting 'age':", age)  # This will raise an error since 'age' is deleted

# Check again if 'age' is in the local and global namespaces
print("'age' in locals():", 'age' in locals())
print("'age' in globals():", 'age' in globals())

print("\n")
# Set the variable 'age' to the value None
age = None
print("After setting 'age' to None:", age)

# Check again if 'age' is in the local and global namespaces
print("'age' in locals():", 'age' in locals())
print("'age' in globals():", 'age' in globals())
```

```
Initial value of 'age': 16
'age' in locals(): True
'age' in globals(): True
'age' in locals(): False
'age' in globals(): False


After setting 'age' to None: None
'age' in locals(): True
'age' in globals(): True
```

-- Unpacking operator

In [47]:
```
def position(x, y, z):
    print(f'Character to {x} {y} {z}')

pos = [5, 10, 15]
position(*pos)
```

```
Character to 5 10 15
```

-- Using "all" operator instead of "and"

In [51]:
```
age = 21
reputation = 20

conditions = [
    age >= 21,
    reputation > 25
]

if all(conditions):
    print("You're an admin. Allowed the access.")
else:
    print("You're a standard user. Not allowed the access.")
```

```
You're a standard user. Not allowed the access.
```

-- Using "any" operator instead of "or"

In [53]:
```
age = 21
reputation = 20

conditions = [
    age >= 21,
    reputation > 25
]

if any(conditions):
    print("You're an admin. Allowed the access.")
else:
    print("You're a standard user. Not allowed the access.")
```

```
You're an admin. Allowed the access.
```

B. Working with String

-- Printing Colored Text

In [7]:
```
print(f"\033[97mCoding is \033[92mexciting")
print(f"\033[97mCoding is \033[92mcreative")
print(f"\033[97mCoding is \033[91mchallenging")
print(f"\033[97mCoding is \033[91mstressful")
print("\n")
print(f"\033[93mCoding Everyday!")
```

```
Coding is exciting
Coding is creative
Coding is challenging
Coding is stressful


Coding Everyday!
```

-- Open a Web Browser

In [13]:
```
import webbrowser

webbrowser.open('https://www.google.com')
```

Out[13]:
```
True
```

-- Concatenation without "+" Operator

In [15]:
```
message = "Hello, this " "string concatenation" " without using '+' sign."
print(message)
```

```
Hello, this string concatenation without using '+' sign.
```

-- "split" Function of string object

In [54]:
```
full_name = "Analytical_Nikita.io"
first_name , last_name = full_name.split(sep= "_")
print(first_name, last_name)
```

```
Analytical Nikita.io
```

-- "join" Function of string object

In [56]:
```
names = ["Analytical", "Nikita.io"]
full_name = "_".join(names)
print(full_name)
```

```
Analytical_Nikita.io
```

-- Working with "in" Substring

In [1]:
```
if "Analytical" in "AnalyticalNikita.io":
    print("The substring is present at", "AnalyticalNikita.io".index("Analytical"), "ind
```

```
The substring is present at 0 index.
```

Note: If the value is not present in the substring it will throw error. For this reason we use find() funtion.

-- Get index using find() method

In [3]:
```
print("The substring is present at", "AnalyticalNikita.io".find("Analytics"), "index.")
```

```
The substring is present at -1 index.
```

-- Using "id" to get identity of the data

Guarenteed to be unique for each object

In [5]:
```
data = {"AnalyticalNikita.io": 1}
print("Address of this data is: ", id(data))
```

```
Address of this data is:  2615849709376
```

-- Aliases

We use an alias if we want two variables pointing to the same data or if we want functions to be able to modify arguments passed it.

In [6]:
```
data1 =  {"AnalyticalNikita.io": 1}
data2 = data1
data2['DataAnalytics'] = 9

print("Address of data1 is: ",id(data1))
print("Address of data2 is: ",id(data2))
print("Data1 now is: ", data1)
```

```
Address of data1 is:  2615850448448
Address of data2 is:  2615850448448
```

```
Data1 now is:  {'AnalyticalNikita.io': 1, 'DataAnalytics': 9}
```

-- Using print end = " " to change ending

In [22]:
```python
favorite_technologies = ["Python", "SQL", "Power BI", "Tableau", "SAS", "Alterxy"]

for technology in (favorite_technologies):
    print(technology, end = " ")

print("") #to go to next line for the next output
```

```
Python SQL Power BI Tableau SAS Alterxy
```

-- Print multiple elements with commas

In [19]:
```python
name = "AnalyticalNikita.io"
favorite_technologies = ["Python", "SQL", "Power BI", "Tableau", "SAS", "Alterxy"]

print(name, "is proficient in", favorite_technologies)
```

```
AnalyticalNikita.io is proficient in ['Python', 'SQL', 'Power BI', 'Tableau', 'SAS', 'Alte
rxy']
```

-- String formatting using "f-string"

In [23]:
```python
name = "AnalyticalNikita.io"

print(f"My name is {name}.") #implict string conversion
```

```
My name is AnalyticalNikita.io.
```

-- Returning multiple values and Assigning to multiple variables

In [26]:
```python
def returning_position():
    #get from user or something
    return 5, 10, 15, 20

print("A tuple", returning_position())
x, y, z, a = returning_position()
print("Assigning to multiple variables: ", "x is", x, "y is", y,"z is", z,"a is", a)
```

```
A tuple (5, 10, 15, 20)
Assigning to multiple variables:  x is 5 y is 10 z is 15 a is 20
```

-- Ternary conditional operator or string comprehension

In [27]:
```python
# Method 1: If - else
reputation = 30
if reputation > 25:
    name = "admin"
else:
    name = "visitor"
print(name)

# Method 2: String comprehension
reputation = 20
name = "admin" if reputation > 25 else "visitor"
print(name)
```

```
admin
visitor
```

-- Flag variable:

Break

In [28]:
```
fruits = ['apple', 'banana', 'orange', 'grape', 'kiwi', 'apple']

# Option 1: Check if 'orange' is in the list directly
if 'orange' in fruits: print("Yes, 'orange' is in the list.")

# Option 2: Use a flag variable to check for 'kiwi'
kiwi_found = False # Assume
for fruit in fruits:
    print(fruit, end=" ") # do something with each element if needed
    if fruit == 'kiwi':
        kiwi_found = True # could count elements
        break

if kiwi_found: print("\nYes, 'kiwi' is in the list.")

# Option 3: Check if 'grapefruit' is in the list without using break
for fruit in fruits:
    print(fruit, end=" ")
    if fruit == 'grapefruit':
        break
else: #no break
    print("\nNo 'grapefruit' found in the list.")
```

```
Yes, 'orange' is in the list.
apple banana orange grape kiwi
Yes, 'kiwi' is in the list.
apple banana orange grape kiwi apple
No 'grapefruit' found in the list.
```

-- Remove list duplicates using set

In [30]:
```
fruits = ['apple', 'banana', 'banana', 'banana', 'kiwi', 'apple']
unique_fruits = list(set(fruits))
print("Unique fruits are: ", unique_fruits)
```

```
Unique fruits are:  ['apple', 'kiwi', 'banana']
```

-- Using "in" method instead of complex conditional

It is popularly used while Checking against list of values

In [40]:
```
weather = "rainy"

if weather in ['rainy', 'cold', 'snowy']:
    print("Plan is cancelled")
```

```
Plan is cancelled
```

C. Working with List/s

-- Reverse a List with Slicing

In [16]:
```
data = [1, 2, 3, 4, 5]
print("Reversed of the string : ", data[::-1])
```

```
Reversed of the string :  [5, 4, 3, 2, 1]
```

-- Reverse a List using reverse() method

In [21]:
```
data = [1, 2, 3, 4, 5]
data.reverse()
print("Reversed of the string : ", data)
```

```
Reversed of the string :  [5, 4, 3, 2, 1]
```

-- in-place methods

Python operate "IN PLACE", like [].sort and [].reverse -- the algorithm does not use extra space for manipulating the input but may require a small though nonconstant extra space for its operation.

In [13]:
```
sort_data = [3, 2, 1]
print("Address of original list is: ", id(sort_data))
sort_data.sort()
print("Sorted list is: ", sort_data)
print("Address of Sorted list is: ", id(sort_data))
sort_data.reverse()
print("Reversed list is: ", sort_data)
print("Address of Reversed  list is: ", id(sort_data))
```

```
Address of original list is:  2615849898048
Sorted list is:  [1, 2, 3]
Address of Sorted list is:  2615849898048
Reversed list is:  [3, 2, 1]
Address of Reversed  list is:  2615849898048
```

Hence all three addresses are same.

-- Replacing "list" vs Replacing list's "content"

In [17]:
```
# Modifies argument pass in (Replace the list with new list)
def replace_list(data):
    """
    The function `replace_list(data)` is creating a new local variable `data` and assign
    This does not modify the original list `programming_languages` that was passed as an
    """
    data = ['Programming Languages']

# Doesn't Modifies argument pass in (Change the data of the list)
def replace_list_content(data):
    """
    The function `replace_list_content` is modifying the content of the list passed as a
    It uses the slice assignment `data[:]` to replace all the elements in the list with
    This means that the original list `programming_languages` will be modified and will
    """
    data[:] = ['Programming Languages']

    # If you need to keep same id just use slicing


programming_languages = ['C', 'C++', 'JavaScript', 'Python']
```

```
programming_languages = [ 'C' ,  'C++' ,  'JavaScript' ,  'Python' ]

print("Original list of languages is:", programming_languages)

# When function modifies the passed-in argument:
replace_list(programming_languages)
print("Modified list of languages is:", programming_languages)

# When function modifies the content of the passed-in argument:
replace_list_content(programming_languages)
print("Unmodified list of languages is:", programming_languages)
```

```
Original list of languages is: ['C', 'C++', 'JavaScript', 'Python']
Modified list of languages is: ['C', 'C++', 'JavaScript', 'Python']
Unmodified list of languages is: ['Programming Languages']
```

-- Copying a List using "Slicing"

In [2]:

```
# Do not point to same exact list address in the memory even if it is copied
programming_languages = ['C', 'C++', 'JavaScript', 'Python']
learning_programming_lamnguages = programming_languages[:]

print("Id of 'programming_languages' is :", id(programming_languages), "\n" "Id of 'lear
```

```
Id of 'programming_languages' is : 1899836578560
Id of 'learning_programming_languages' is : 1899836579328
```

-- Copy a List using "copy()" method

Return a shallow copy of 'programming_language'. If there is any Nested objects just the reference to those objects is what copied (address) and the data object iself is not copied.

In [3]:

```
programming_languages = ['C', 'C++', 'JavaScript', 'Python']
learning_programming_lamnguages = programming_languages.copy()

print("Id of 'programming_languages' is :", id(programming_languages), "\n" "Id of 'lear
```

```
Id of 'programming_languages' is : 1899836614272
Id of 'learning_programming_languages' is : 1899836577536
```

-- Copying a List using "deepcopy()"

A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

In [4]:

```
import copy

original_list = [1, [2, 3], [4, 5]]
shallow_copied_list = copy.copy(original_list)
deep_copied_list = copy.deepcopy(original_list) # To use 'deepcopy' : from copy import d

# Modify the original list
original_list[1][0] = 'X'

print("Original List:", original_list)
print("Shallow Copied List:", shallow_copied_list)
print("Deep Copied List:", deep_copied_list)
```

```
Original List: [1, ['X', 3], [4, 5]]
Shallow Copied List: [1, ['X', 3], [4, 5]]
Deep Copied List: [1, [2, 3], [4, 5]]
```

-- Concatenating lists using "+" Operator

In [5]:
```
x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]

print("The concatenated list is:", x + y + z)
```

```
The concatenated list is: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

-- Comparing "not" vs "is None"

In [12]:
```
data1 = []
data2 = None
print("Is data1 empty:", not data1)
print("Is data1 None:", data1 is None)


# None evaluates to false
print("Is data2 Empty:", not data2) # `not data2` is checking if `data2` is empty or eva
print("Is data2 None:",data2 is None)
```

```
Is data1 empty: True
Is data1 None: False
Is data2 Empty: True
Is data2 None: True
```

-- Checking a list is empty or none

"Not data" work whether it's "None" or "empty" list

In [9]:
```
data = None # []

print("None or empty list:", not data) #empty, None, or zero

#You can also check these things:
print("Anything but None:", data is not None)
print("None only:", data is None)
```

```
None or empty list: True
Anything but None: False
None only: True
```

-- Use range to generate lists

In [44]:
```
print(list(range(0, 10, 3)))
print(type(range(0, 10, 3))) # python 3
```

```
[0, 3, 6, 9]
<class 'range'>
```

-- List comprehension (Removing certain elements from a list) instead of "for row" loops

In [57]:
```
fruits = ['apple', 'banana', 'orange', 'grape', 'kiwi', 'apple']

removing_fruits = [fruit for fruit in fruits if fruit not in ["kiwi", "apple"]]
```

```
print("Fruits are: ", removing_fruits)
```

```
Fruits are:  ['banana', 'orange', 'grape']
```

-- List "reversed" method

In [63]:

```
fruits = ['apple', 'banana', 'orange', 'grape', 'kiwi', 'apple']
for pet in reversed(fruits):
    print(pet, end=" ")
```

```
apple kiwi grape orange banana apple
```

-- "flatten" a list of nested lists

In [5]:

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


# Method 1: Using Nested Loop:

flattened_list = []
for sublist in nested_list:
    for item in sublist:
        flattened_list.append(item)

print("Flattened List (using nested loop):", flattened_list)


# Method 2: Using List Comprehension:
flattened_list = [item for sublist in nested_list for item in sublist]

print("Flattened List (using list comprehension):", flattened_list)

# Method 3 : Using Recursive Function:
def flatten_list(nested_list):
    flattened_list = []
    for item in nested_list:
        if isinstance(item, list):
            flattened_list.extend(flatten_list(item))
        else:
            flattened_list.append(item)
    return flattened_list

nested_list = [[1, 2, 3], [4, 5], [6, 7, [8, 9]]]
flattened_list_recursive = flatten_list(nested_list)

print("Flattened List (using recursive function):", flattened_list_recursive)
```

```
Flattened List (using nested loop): [1, 2, 3, 4, 5, 6, 7, 8]
Flattened List (using list comprehension): [1, 2, 3, 4, 5, 6, 7, 8]
Flattened List (using recursive function): [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

-- Space separated numbers to integer list

In [30]:

```
user_input = "1 2 3 4 5"
new_list = list(map(int, user_input.split()))
print("The list: ", new_list)
```

```
The list:  [1, 2, 3, 4, 5]
```

-- Combine two lists as a list of lists

-- Combine two lists as a list of lists

In [18]:
```
# Different set of names and points
names = ['Alice', 'Bob', 'Eva', 'David']
points = [80, 300, 50, 450]

zipped = list(zip(names, points))

print(zipped)
```

```
[('Alice', 80), ('Bob', 300), ('Eva', 50), ('David', 450)]
```

-- Convert list of tuples to list of lists

In [19]:
```
# Different set of names and points
names = ['Alice', 'Bob', 'Eva', 'David']
points = [80, 300, 50, 450]

zipped = list(zip(names, points))

data = [list(item) for item in zipped]
print(data)
```

```
[['Alice', 80], ['Bob', 300], ['Eva', 50], ['David', 450]]
```

D. General

-- "do-while" loop in python

In [1]:
```
while True:
    print("""Choose an option:
1. play game
2. load game
3. high scores
4. quit""")

    # if input() == "4":
    if True:
        break
```

```
Choose an option:
1. play game
2. load game
3. high scores
4. quit
```

-- Doing nothing in Python with "pass"

In [49]:
```
def positional_variables():
    pass
```

-- enumerate() function insted of range(len())

In [3]:
```
fruits = ['apple', 'banana', 'kiwi', 'orange']
```

```
# Using enumerate to iterate over the list with both index and value
for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")

print("\n")
# You can also specify a start index (default is 0)
for index, fruit in enumerate(fruits, start=1):
    print(f"Index {index}: {fruit}")
```

```
Index 0: apple
Index 1: banana
Index 2: kiwi
Index 3: orange


Index 1: apple
Index 2: banana
Index 3: kiwi
Index 4: orange
```

-- Assigning a function to a variable - just don't use ()

In [7]:

```
def done():
    print("done")

def do_something(callback):
    print("Doing things....") # callback function as an argument and prints "Doing thing

    """
    callback()  # Call the provided callback function

    # to print output
    Doing things....
    done
    """

# Call do_something with the done function as the callback
do_something(done)
```

```
Doing things....
```

-- Wait with time.sleep

In [9]:

```
import time

def done():
    print("done")

def do_something(callback):
    time.sleep(2) # it will print output after some time for ex 2 means 2.0s
    print("Doing things....") # callback function as an argument and prints "Doing thing
    callback() # Call the provided callback function


# Call do_something with the done function as the callback
do_something(done)
```

```
Doing things....
done
```

-- Sort complex iterables with sorted()

In [10]:
```
dictionary_data = [{"name": "Max", "age": 6},
                   {"name": "Max", "age": 61},
                   {"name": "Max", "age": 36},
                   ]

sorted_data = sorted(dictionary_data, key=lambda x : x["age"])
print("Sorted data: ", sorted_data)
```

```
Sorted data:  [{'name': 'Max', 'age': 6}, {'name': 'Max', 'age': 36}, {'name': 'Max', 'ag
e': 61}]
```

-- Save memory with "Generators"

In [14]:
```
import sys

my_list = [i for i in range(1000)]
print(sum(my_list))
print("Size of list", sys.getsizeof(my_list), "bytes")


my_gen = (i for i in range(1000))
print(sum(my_gen))
print("Size of generator", sys.getsizeof(my_gen), "bytes")
```

```
499500
Size of list 8856 bytes
499500
Size of generator 112 bytes
```

-- Using zip:

In [11]:
```
# Example using zip with two lists
numbers = [1, 2, 3]
letters = ['a', 'b', 'c']

# Zip combines corresponding elements from both lists
zipped_result = zip(numbers, letters)

# Iterate over the zipped result
for number, letter in zipped_result:
    print(f"Number: {number}, Letter: {letter}")
```

```
Number: 1, Letter: a
Number: 2, Letter: b
Number: 3, Letter: c
```

-- Using zip_longest from itertools:

In [12]:
```
from itertools import zip_longest

# Example using zip_longest with two lists of different lengths
numbers = [1, 2, 3]
letters = ['a', 'b']

# zip_longest fills missing values with a specified fillvalue (default is None)
zipped_longest_result = zip_longest(numbers, letters, fillvalue='N/A')

# Iterate over the zipped_longest result
for number, letter in zipped_longest_result:
    print(f"Number: {number}, Letter: {letter}")
```

```
Number: 1, Letter: a
```

```
Number: 2, Letter: b
Number: 3, Letter: N/A
```

-- Default Arguments

When you have default values, you can pass arugments by name positional arguments must remain on the left

In [20]:
```
from itertools import zip_longest


def zip_lists(list1=[], list2=[], longest=True):
    if longest:
        return [list(item) for item in zip_longest(list1, list2)]
    else:
        return [list(item) for item in zip(list1, list2)]

names = ['Alice', 'Bob', 'Eva', 'David', 'Sam', 'Ace']
points = [100, 250, 30, 600]

print(zip_lists(names, points))
```

```
[['Alice', 100], ['Bob', 250], ['Eva', 30], ['David', 600], ['Sam', None], ['Ace', None]]
```

-- Keyword Arguments

You can pass named arguments in any order and can skip them even.

In [22]:
```
from itertools import zip_longest


def zip_lists(list1=[], list2=[], longest=True):
    if longest:
        return [list(item) for item in zip_longest(list1, list2)]
    else:
        return [list(item) for item in zip(list1, list2)]


print(zip_lists(longest=True, list2=['Eva']))
```

```
[[None, 'Eva']]
```

-- Get the Python version

In [13]:
```
from platform import python_version

print(python_version())
```

```
3.9.13
```

-- Define default values in Dictionaries with .get() and .setdefault()

In [17]:
```
my_dict = {"name": "Max", "age": 6}
count = my_dict.get("count")
print("Count is there or not:", count)

# Setting default value if count is none
count = my_dict.setdefault("count", 9)
```

```
print("Count is there or not: ", count)
print("Updated my_dict:", my_dict)
```

```
Count is there or not: None
Count is there or not: 9
Updated my_dict: {'name': 'Max', 'age': 6, 'count': 9}
```

-- Using "Counter" from collections

In [22]:

```
from collections import Counter

my_list = [1,2,1,2,2,2,4,3,4,4,5,4]
counter = Counter(my_list)
print("Count of the numbers are: ", counter)

most_commmon = counter.most_common(2) # passed in Number will denotes how many common nu
print("Most Common Number is: ", most_commmon[0]) # printin zeroth index element from 2
```

```
Count of the numbers are:  Counter({2: 4, 4: 4, 1: 2, 3: 1, 5: 1})
Most Common Number is:  (2, 4)
```

-- Merging two dictionaries using **

In [23]:

```
d1 = {"name": "Max", "age": 6}
d2 = {"name": "Max", "city": "NY"}

merged_dict = {**d1, **d2}
print("Here is merged dictionary: ", merged_dict)
```

```
Here is merged dictionary:  {'name': 'Max', 'age': 6, 'city': 'NY'}
```

-- Get current date and time

In [24]:

```
from datetime import datetime
now = datetime.now()
print(now)
print(now.day, now.month, now.year, now.hour, now.minute, now.second)
```

```
2023-11-22 22:23:49.672825
22 11 2023 22 23 49
```

-- Countdown

You can do this in a loop if you need.

In [26]:

```
now = datetime.now()
end = datetime(2020, 12, 1)
print(end-now)
```

```
-1087 days, 1:34:50.201422
```

-- Elapsed time

In [27]:

```
start = datetime.now()

for i in range(100_000_000): # to pass time
    pass
```

```
end = datetime.now()

print(type(end-start))
elapsed = end-start
print(elapsed)
print(elapsed.seconds, elapsed.microseconds)
```

```
<class 'datetime.timedelta'>
0:00:02.690219
2 690219
```

-- Parentheses for operations with object members

operator precedence - dot operator comes ahead of +/- operator.

In [28]:

```
now = datetime.now()
then = datetime.now()

elapsed = (then-now).microseconds
print(elapsed)

try:
    print(then-now.microseconds)
except:
    print("Wrong")
```

```
0
Wrong
```

-- Runtime error vs syntax error

Syntax errors are impossible to be correct and will prevent execution

Runtime errors deal with incorrect data found during runtime

-- Get a random number

In [29]:

```
from random import randint
print(randint(0, 12)) #inclusive #inclusive
```

```
7
```

-- Class Parenthesis Optional but Not Function Parenthesis

- It shows two ways to define a class: class Container(): and class Container:.
- The parentheses after Container are optional for classes, but they are needed if you are inheriting from another class.
- In contrast, function parentheses are always required when defining a function.

In [7]:

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

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

-- Wrapping a Primitive to Change Within a Function

Container is a simple class that wraps a primitive (in this case, an integer).

In [1]:
```
# The code defines a class called `Container` with a constructor method `__init__`
# that takes a parameter `data` and assigns it to an instance variable `self.data`.
class Container:
    def __init__(self, data):
        self.data = data

def calculate(input):
    input.data **= 5

container = Container(5)
calculate(container)
print(container.data)
```

```
3125
```

-- Compare Identity with "is" operator

In [3]:
```
c1 = Container(5)
c2 = Container(5)

print(id(c1), id(c2))
print(id(c1) == id(c2))  # returns False because they are different objects.
print(c1 is c2) # same objects but returns False because they are distinct instances.
```

```
1274963509840 1274946106128
False
False
False
```

-- Add a Method Dynamically

- A custom **eq** method is defined outside the class.
- This method is then added to the Container class dynamically using Container.**eq** = **eq**.
- The print(Container.**eq**) statement shows the memory address of the **eq** method.

In [5]:
```
def __eq__(self, other):
    return self.data == other.data

Container.__eq__ = __eq__

print(Container.__eq__)
```

```
<function __eq__ at 0x00000128D8FE04C0>
```

-- Now Compare by Value

The **eq** method added dynamically in the previous step is used for the equality check (c1 == c2).

In [6]:
```
c1 = Container(5)
c2 = Container(5)

print(c1 == c2)  # Compares by value
# This time, the result is True because the custom __eq__ method compares
# the values inside the Container instances.
```

```
print(c1 is c2)  # Compares by identity (address)
# The is operator still returns False because it checks for identity.
```

```
True
False
```

-- Fibonacci Generator and Yield

- fib is a generator function that generates Fibonacci numbers up to a specified count.
- gen is an instance of the generator, and next is used to retrieve the next values.
- The second loop demonstrates using a generator in a for loop to print the first 20 Fibonacci numbers.

In [8]:

```
def fib(count):
    a, b = 0, 1
    while count:
        yield a
        a, b = b, b + a
        count -= 1

gen = fib(100)
print(next(gen), next(gen), next(gen), next(gen), next(gen))

for i in fib(20):
    print(i, end=" ")
```

```
0 1 1 2 3
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

-- Infinite Number and Infinite Generator

- math.inf represents positive infinity.
- The fib generator is used with an infinite loop, and values are printed until the condition i >= 200 is met.

This code demonstrates the use of positive infinity, a Fibonacci number generator, and how to break out of the generator loop based on a condition.

In [10]:

```
import math

# Printing Infinity: special floating-point representation
print(math.inf)

# Assign infinity to a variable and perform an operation
inf = math.inf
print(inf, inf - 1)  # Always infinity, Even when subtracting 1, the result is still inf


# Fibonacci Generator:
def fib(count):
    a, b = 0, 1
    while count:
        yield a
        a, b = b, b + a
        count -= 1

# Using the Fibonacci Generator:
# Use the Fibonacci generator with an infinite count
f = fib(math.inf)

# Iterate through the Fibonacci numbers until a condition is met
```

```
# Iterate through the Fibonacci numbers until a condition is met
for i in f:
    if i >= 200:
        break
    print(i, end=" ")
```

```
inf
inf inf
0 1 1 2 3 5 8 13 21 34 55 89 144
```

-- List from Generator

In [11]:

```
import math

# The code is creating a Fibonacci sequence generator using a generator function called
def fib(count):
    a, b = 0, 1
    while count:
        yield a
        a, b = b, b + a
        count -= 1

# The `fib` function takes a parameter `count` which determines the number of Fibonacci
f = fib(10)

# This code generates Fibonacci numbers and creates a list containing the square root of
data = [round(math.sqrt(i), 3) for i in f]
print(data)
```

```
[0.0, 1.0, 1.0, 1.414, 1.732, 2.236, 2.828, 3.606, 4.583, 5.831]
```

-- Simple Infinite Generator with "itertools"

The generator function could be simpler without having to take a max count property. This can be done easily with itertools.

In [12]:

```
import itertools

def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, b + a

# itertools.islice is used to get the first 20 values from an infinite generator.
print(list(itertools.islice(fib(), 20)))
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]
```

-- Iterate Through Custom Type with **iter**

In [13]:

```
# Defines a custom LinkedList class with a Node class as an element.

class Node:
    def __init__(self, data, next_node=None):
        self.data = data
        self.next = next_node

class LinkedList:
    def __init__(self, start):
        self.start = start
```

```
    # The __iter__ method is implemented to allow iteration over the linked list.
    def __iter__(self):
        node = self.start
        while node:
            yield node
            node = node.next

ll = LinkedList(Node(5, Node(10, Node(15, Node(20)))))
for node in ll:
    print(node.data)
```

```
5
10
15
20
```

-- Falsey for Custom Types (not)

In [14]:

```
# Demonstrates how not works with custom types, considering an empty list as True.
print(not [])  # True because it's empty
print(len([]) == 0)  # This is how `not` is evaluated

# Adds a custom __len__ method to the LinkedList class to get the length of the linked l
def __len__(self):
    count = 0
    for i in self:
        count += 1
    return count

LinkedList.__len__ = __len__
```