SQLabs

המכללה הגבוהה להייטק

Python Programming
for Cyberark
Instructor: David Melnik
Day 1

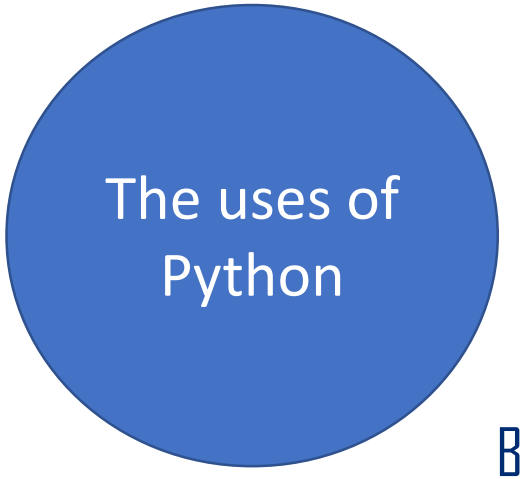
SQLabs
המכללה הגבוהה להייטק

The history of Python

- The creator of Python is Guido van Rossum
- In December 1989, the first version of Python was born
- Python 3.0 was released on December 3, 2008
- The new version was not backward-compatible with the previous version
- Python is one of the most popular programming languages

SQLabs
המכללה הגבוהה להייטק

which professions can use Python?



Software Developers

Data Scientists and Analysts

Machine Learning Engineers

Data Engineers

DevOps Engineers

Network and Systems Administrators

Quality Assurance Engineers

Websites development

Game Developers

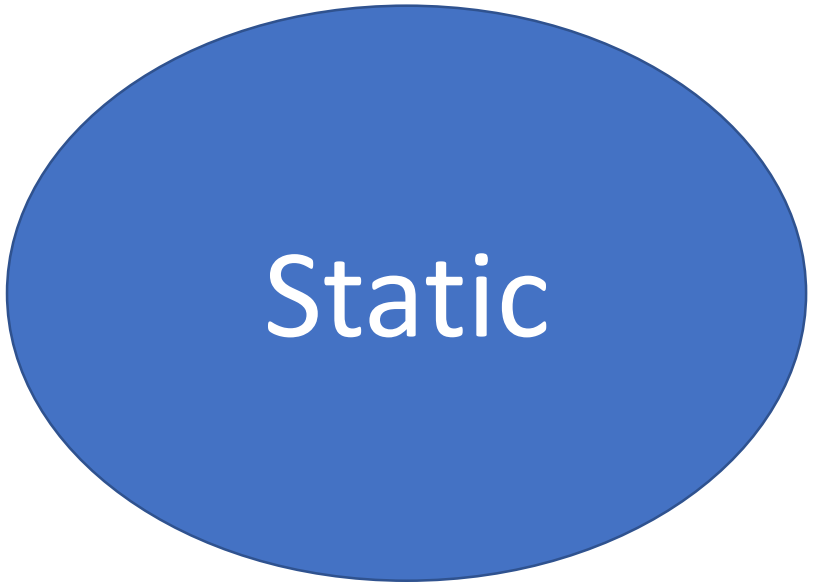
Cybersecurity Professionals

Financial Analysts

Bioinformaticians

SQLabs
המכללה הגבוהה להייטק

Static vs. Dynamic Languages



JAVA

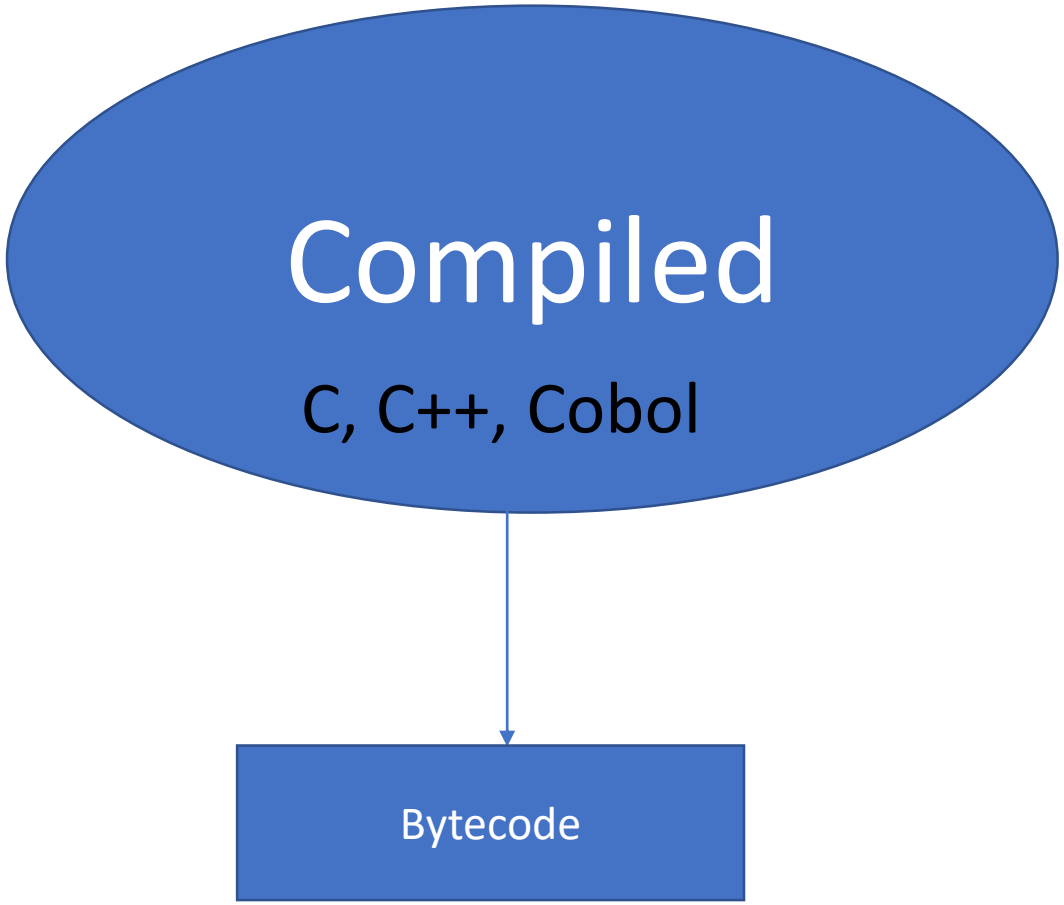
```
int x = 5;
boolean b = x;
```

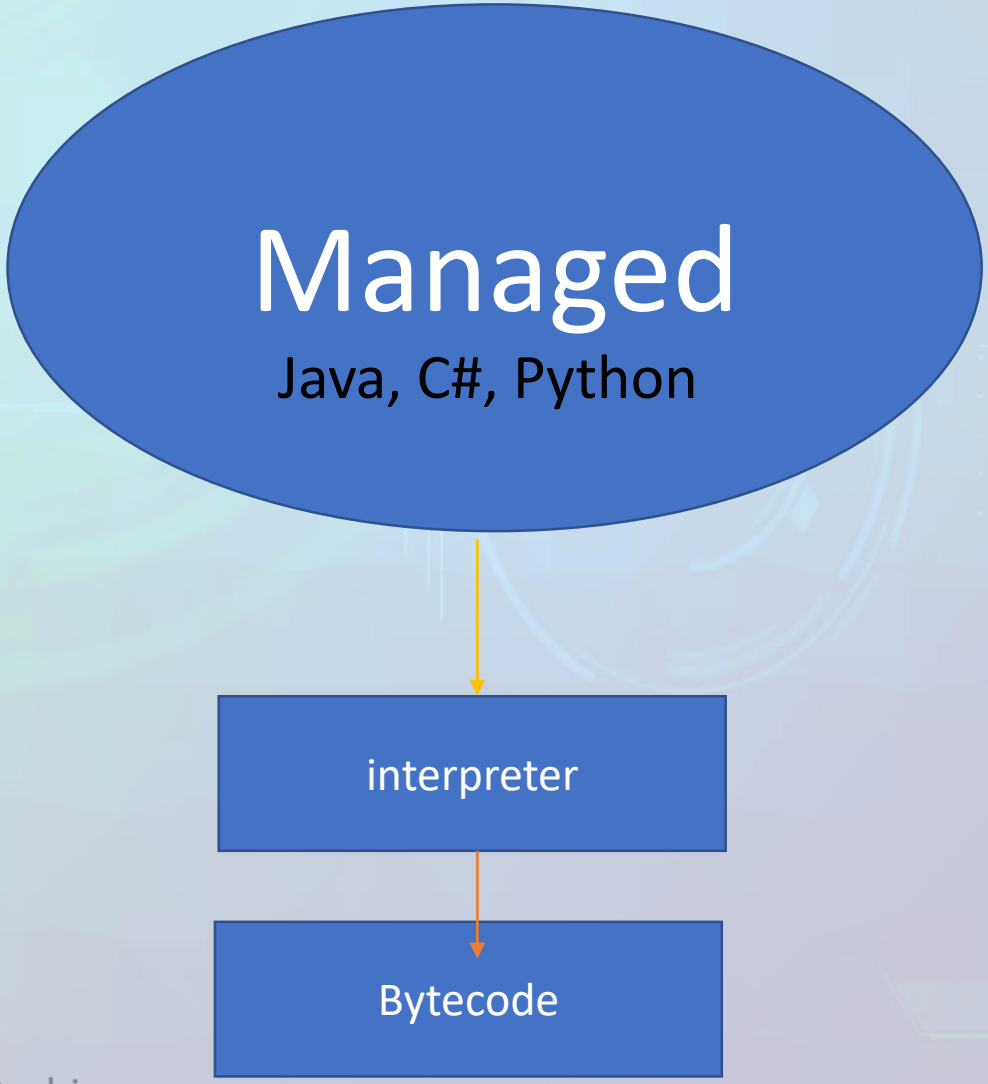
PYTHON

```
x = 5
b = True
b = x
b = “hello”
```

SQLabs
המכללה הגבוהה להייטק

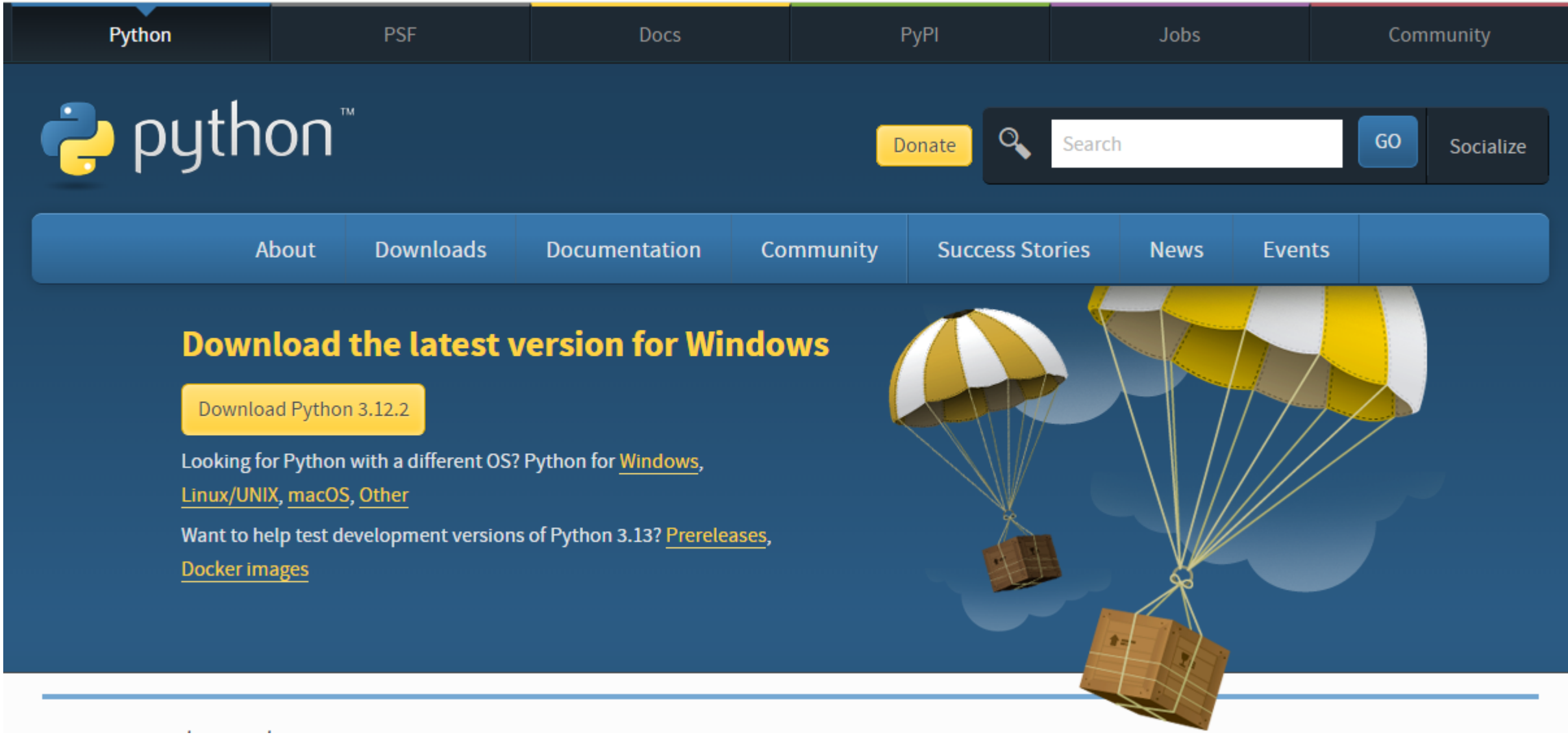
Managed Vs. Compiled Languages



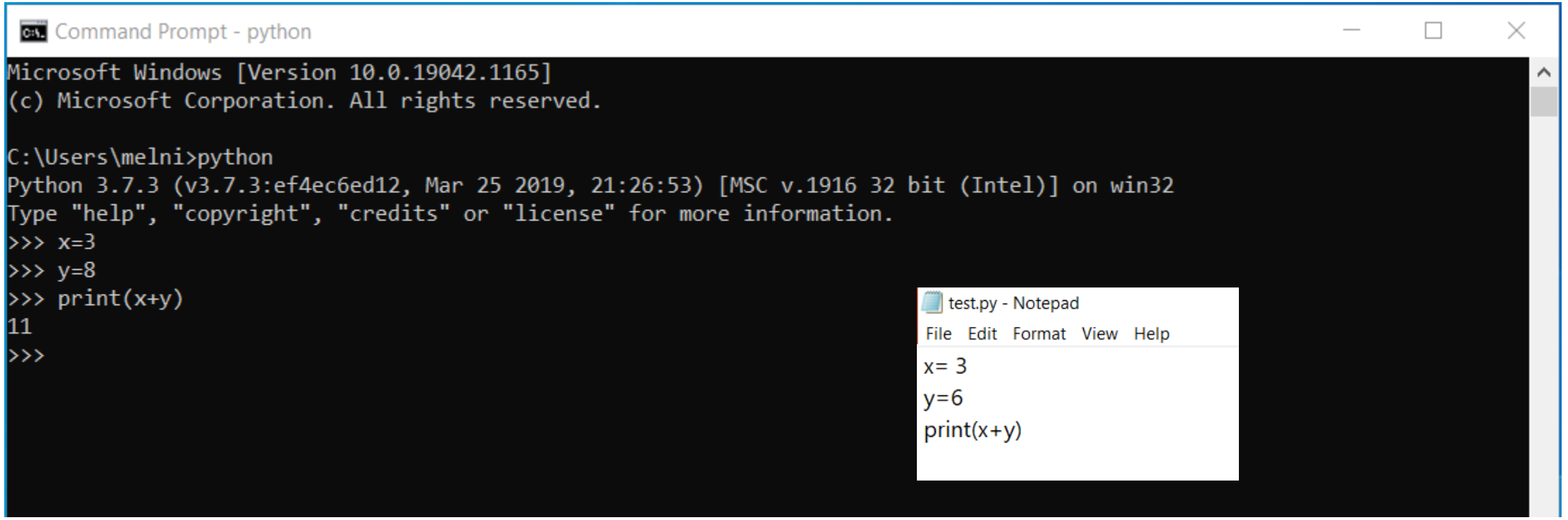


SQLabs
המכללה הגבוהה להייטק

https://www.python.org/downloads/

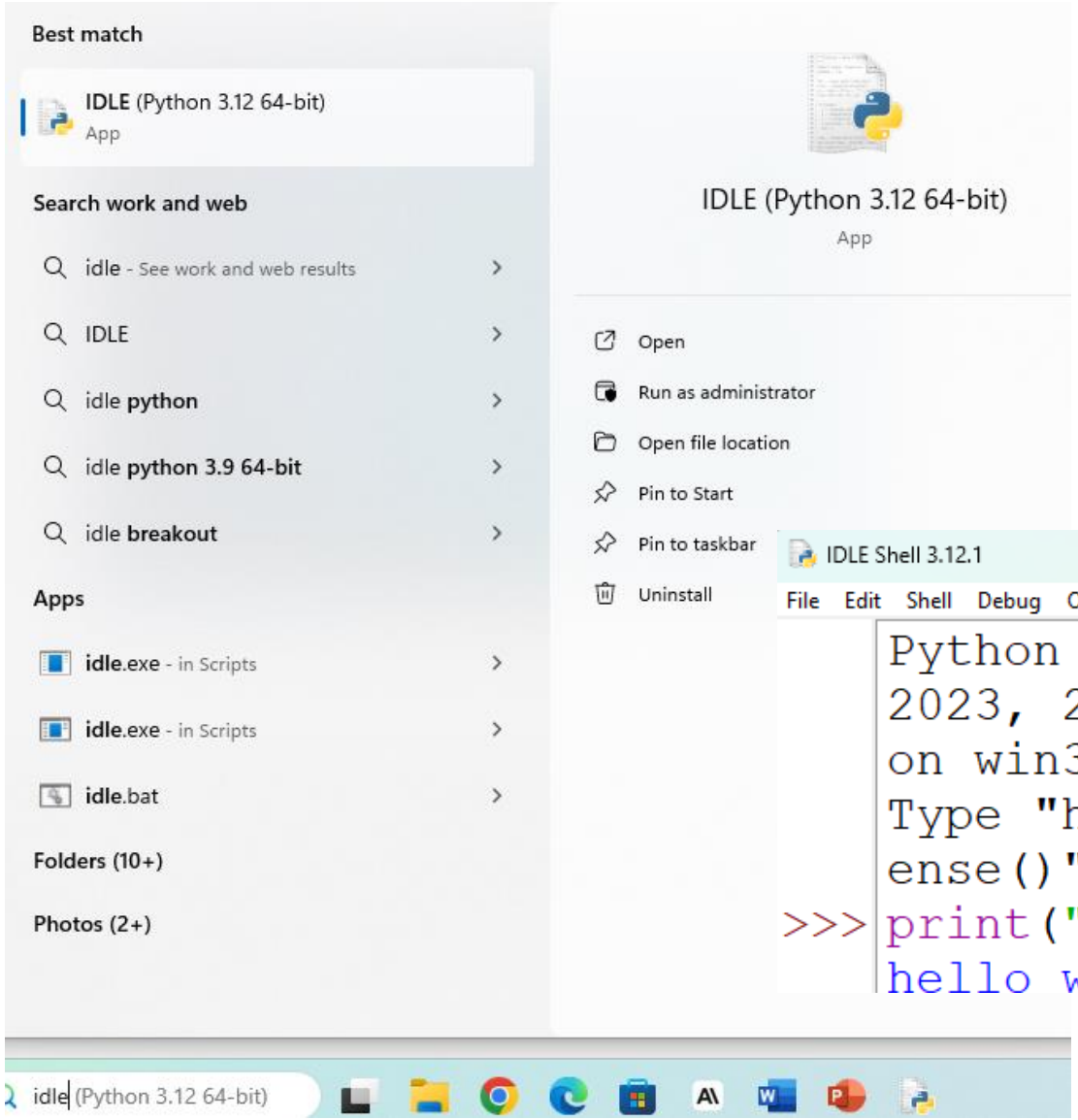


How to run python code



SQLabs
המכללה הגבוהה להייטק

Idle shell



IDLE (Python 3.12 64-bit)

App

Open

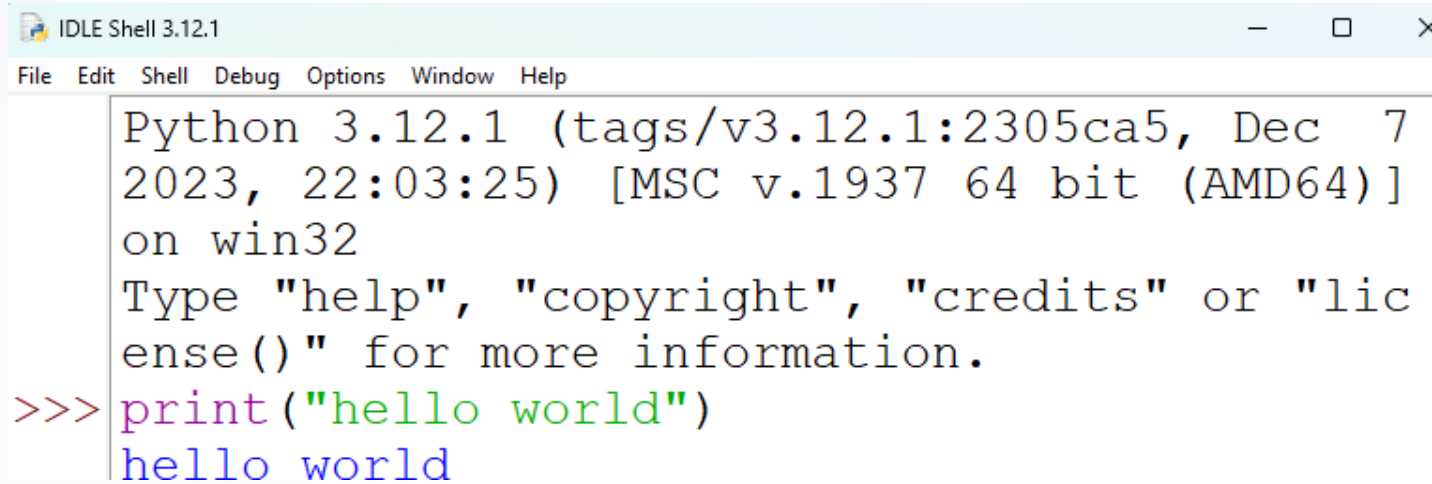
Run as administrator

Open file location

Pin to Start

Pin to taskbar

Uninstall



Development Tools

IDE - Integrated Development Environment

- Save and Reload Source Code
- Execution from Within the Environment
- Debugging Support
- Syntax Highlighting
- Automatic Code Formatting

VS Code

SQLabs
המכללה הגבוהה להייטק

VIRTUAL ENVIRONMENT

A virtual environment is a self-contained directory tree that contains a Python installation for a particular version of Python, plus additional packages.

Dependency Management: Different projects might require different versions of libraries or Python itself.
Consistency Across Environments: Virtual environments help ensure that your project runs consistently across different machines or deployment environments.
System Integrity: you don't have to install packages globally, thus avoiding messing with system-wide Python and its packages.
Simplification of Package Management: Tools like pip work seamlessly with virtual environments, allowing for easy installation, upgrade, and removal of packages without affecting other projects.

Using Virtual Environments

It's a best practice to use virtual environments to manage dependencies for projects.

To create a virtual environment, you can use venv (standard library in Python 3.3 and later)

```
python -m venv myprojectenv
```

To activate the virtual environment:

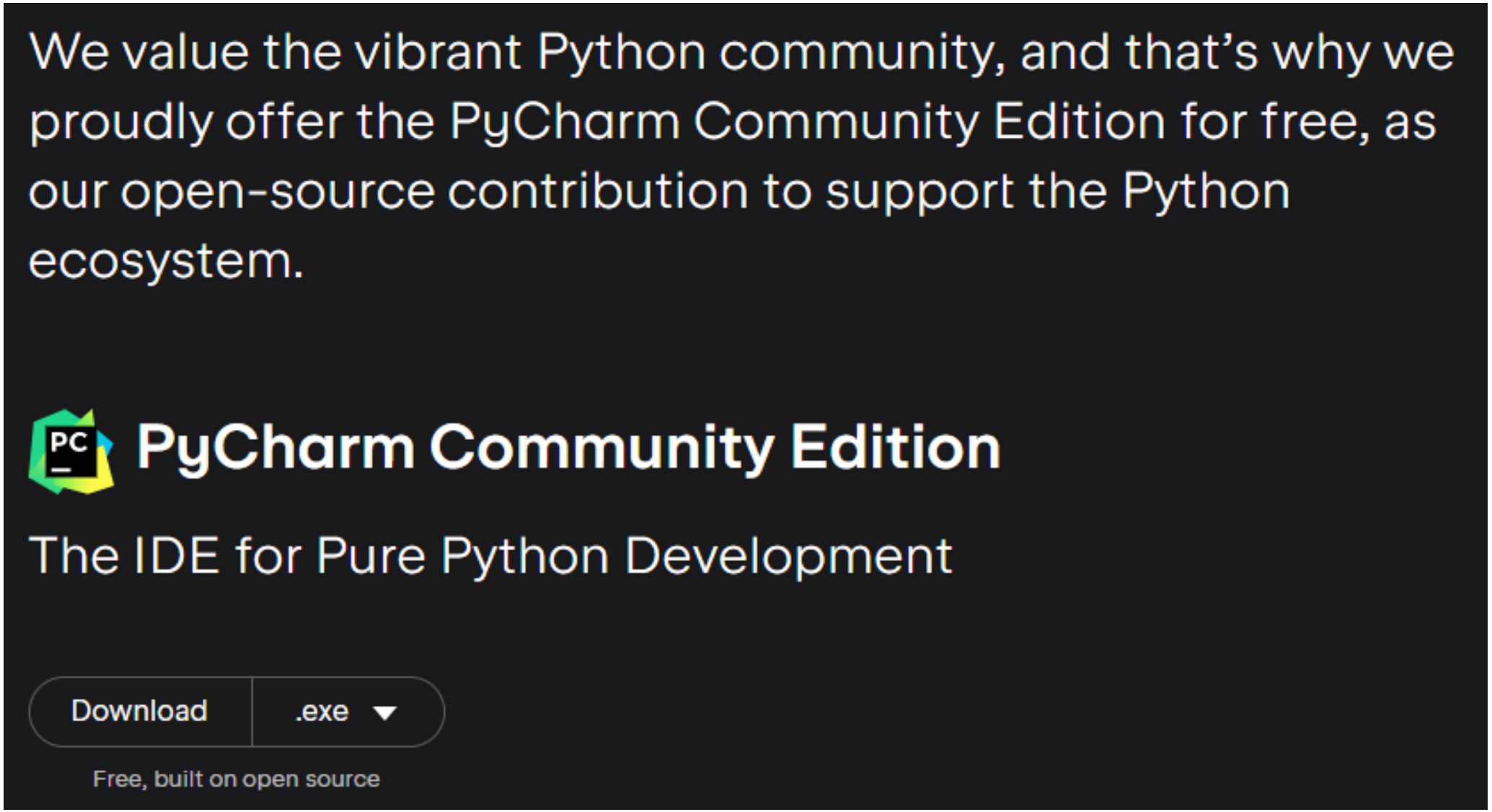
On Windows:

```
myprojectenv\Scripts\activate
```

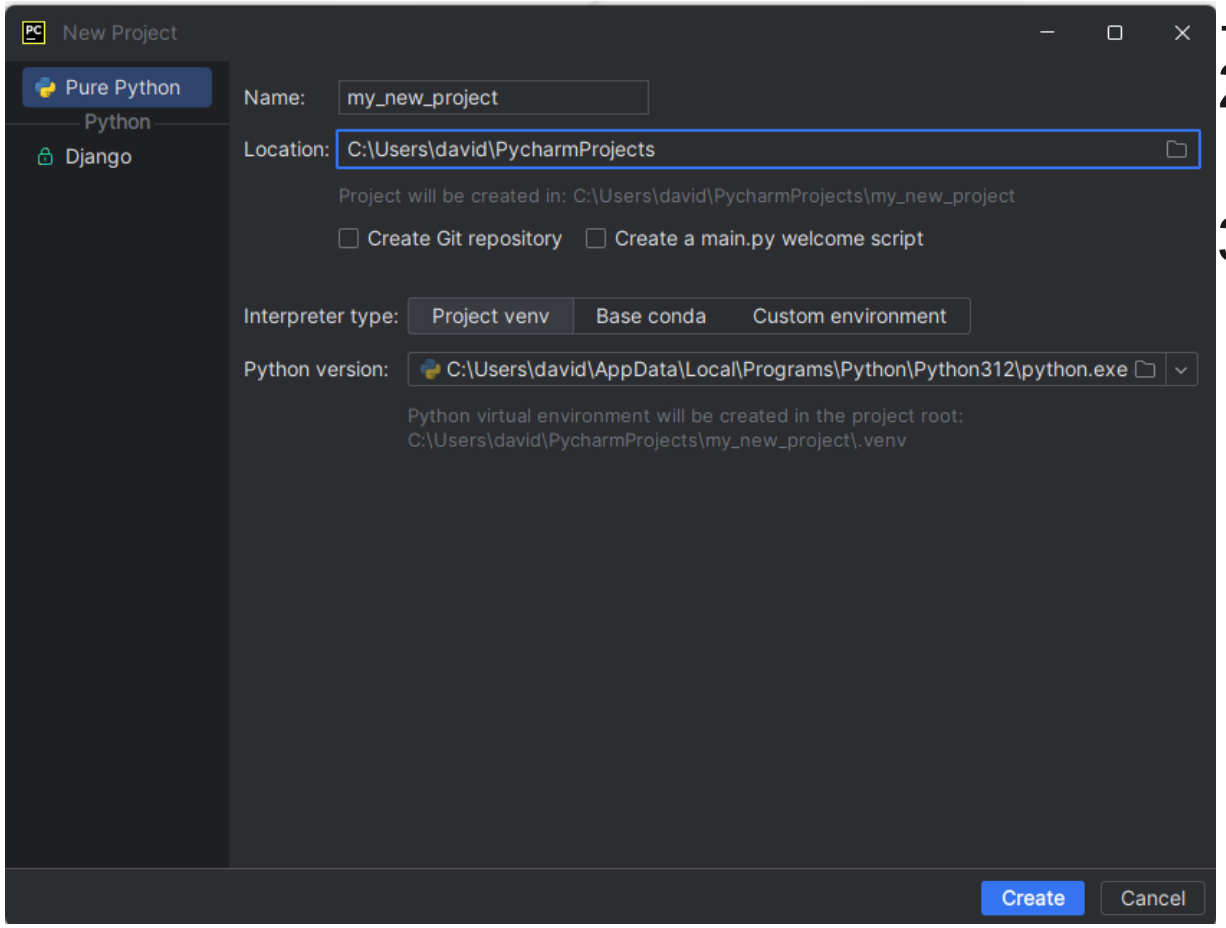
On Unix or MacOS:

```
source myprojectenv/bin/activate
```

https://www.jetbrains.com/pycharm/download



PyCharm – new project



1. Choose the project location .
2. Click in the **Location** dna dleif .tcejorp ruoy rof yrotcerid eht yficeps
3. Python best practice is to create a dedicated environment for each project. In most cases, the default **Project venv** t'now uoy dna ,boj eht od lliw .gnihtyna erugifnoc ot deen

Still, you can switch to **Custom environment** na esu ot elba eb ot rehto tceles ,tnemnorivne gnitsixe eht yficeps ,sepyt tnemnorivne rehto yfidom dna ,noitacol tnemnorivne .snoitpo

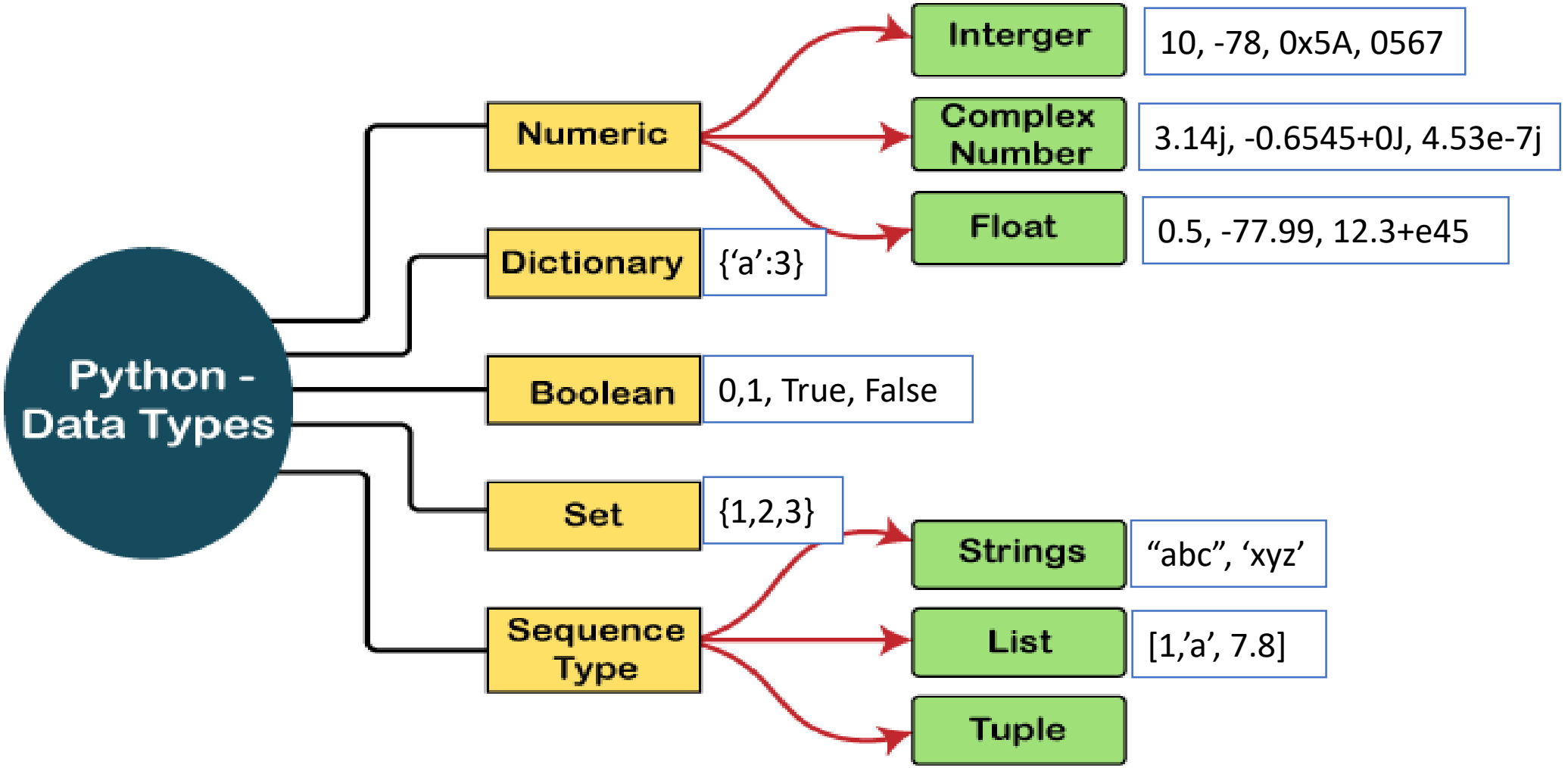
SQLabs
המכללה הגבוהה להייטק

basic python programming

- Python Data Types
- Operators and Methods
- Range, Loops
- Flow Control
- Functions, Args, Kwargs, and Scope

SQLabs
המכללה הגבוהה להייטק

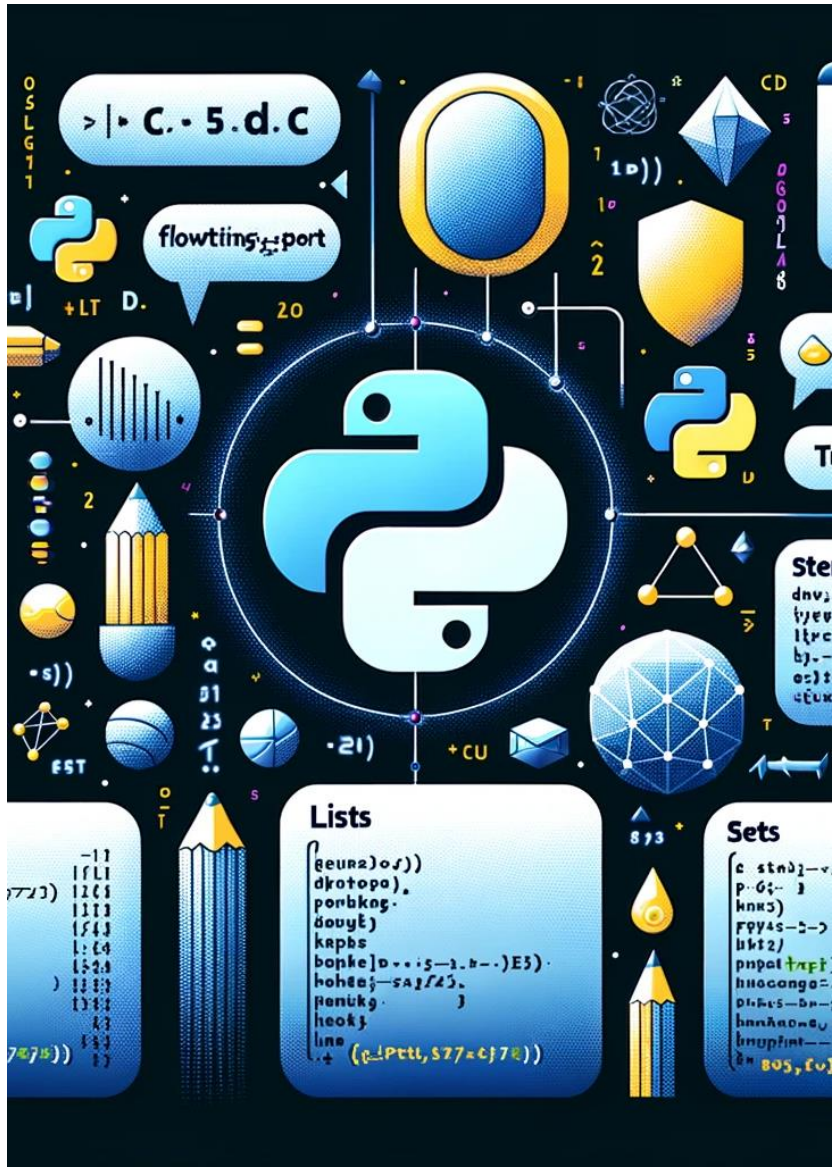
PYTHON DATA TYPES



Mutable VS immutable

In Python, data types can be classified into two categories based on whether they can be changed after their creation: **mutable** and **immutable** types.

Mutable types	Immutable types
can be changed after they are created	cannot be changed after they are created
list, **dict**, **set**.	**Int**, **float**, **bool**, **str**, **tuple**, **Complex**

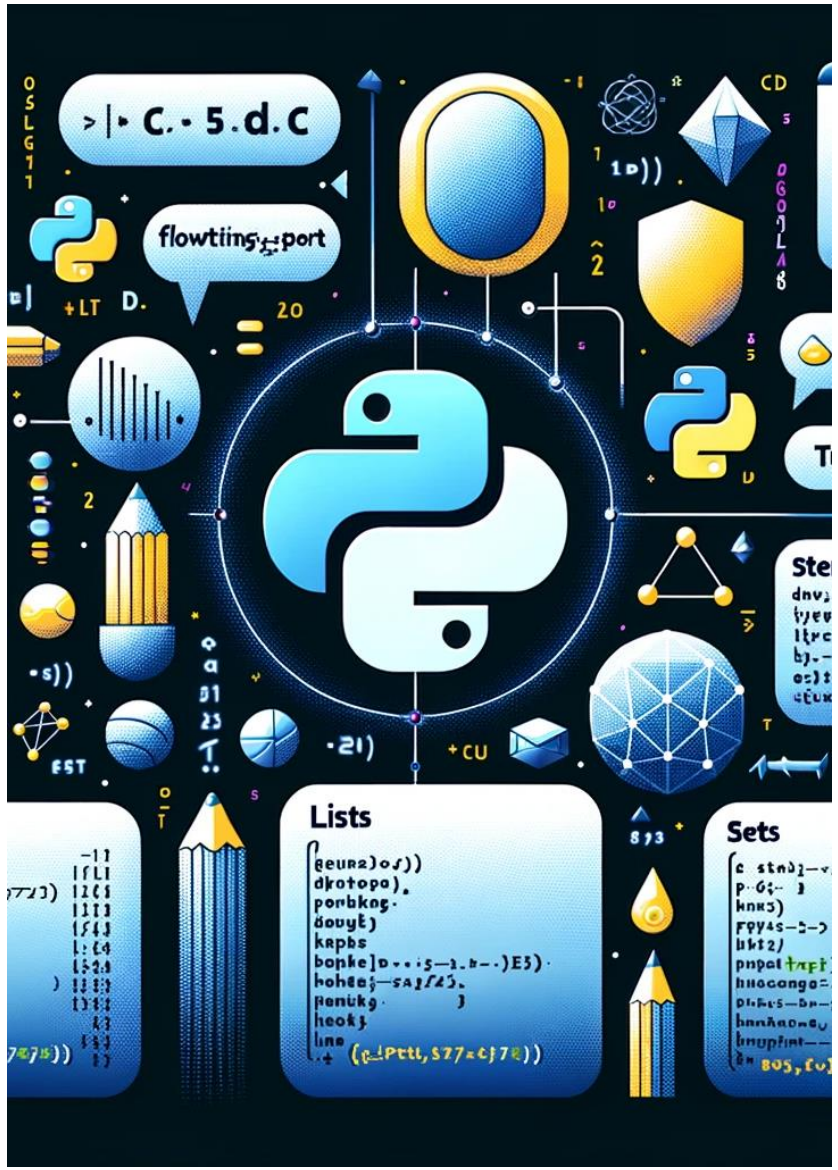


PYTHON DATA TYPES - BOOL

bool (Boolean): Represents **True** or **False** values.

boolean operators are used to perform logical operations, returning either **True** or **False**.

Operator	Description	Example
and	Returns True if both operands are true	x and y
OR	Returns True if at least one of the operands is true	x or y
not	Returns True if the operand is false	not x

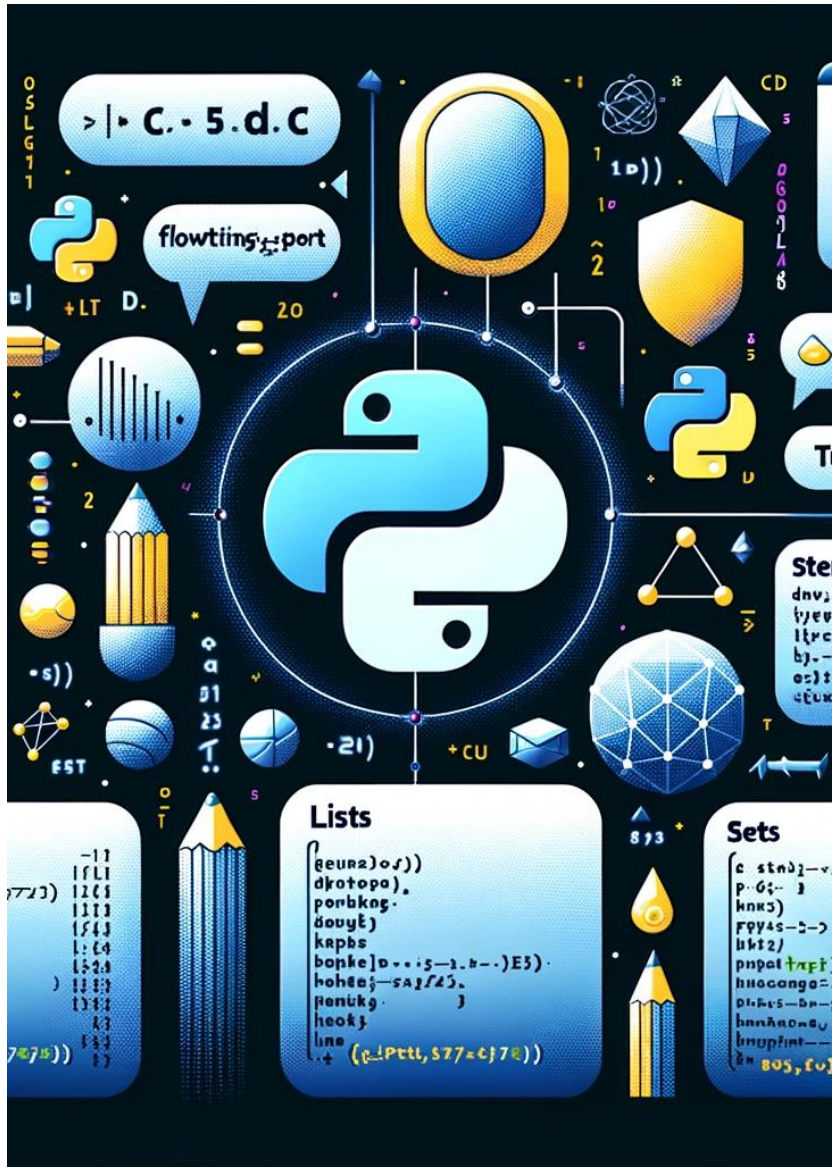


PYTHON DATA TYPES

Bitwise operators

Operator	Description	Example
&	Performs AND on each pair of bits	x & y
|	Performs OR on each pair of bits	x | y
^	Performs XOR on each pair of bits	x ^ y
~	Inverts each bit	~x
>>	Shifts bits to the left	x << n
<<	Shifts bits to the right	x >> n

SQLabs
המכללה הגבוהה להייטק



PYTHON DATA TYPES - NUMERICS

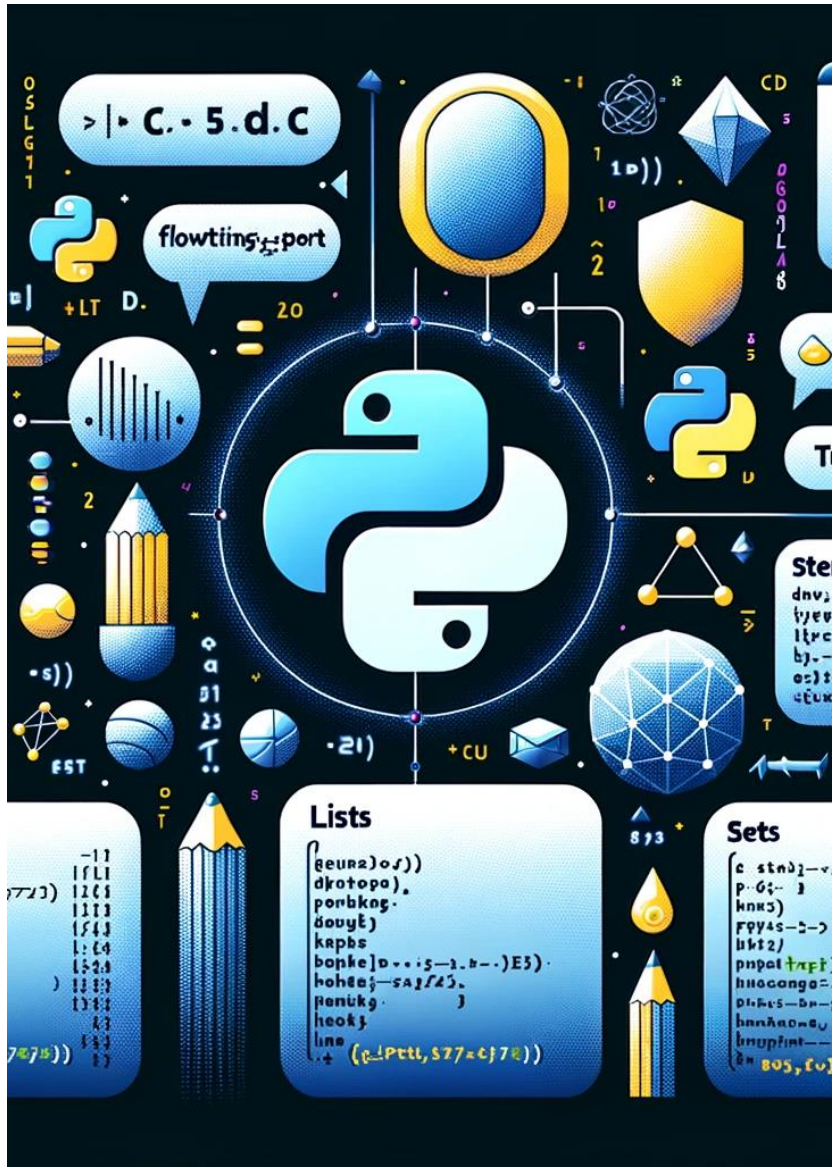
The Numeric Types

- **int** (Integer): Whole numbers, positive or negative, without decimals. Example: 5
- **float** (Floating Point): Numbers that contain a decimal point. Example: 3.14
- **complex** (Complex Numbers): Numbers with a real and imaginary part. Example: 3+5j

PYTHON DATA TYPES - NUMERICS

Numeric Operators

Operator	Description	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
//	Floor Division: Division that results into whole number	x // y
%	Modulus: returns the remainder of the division	x % y
**	Exponentiation: Raises first operand to power of second	x ** y



PYTHON DATA TYPES - NUMERICS

augmented assignment:

x += 2

x -= 2

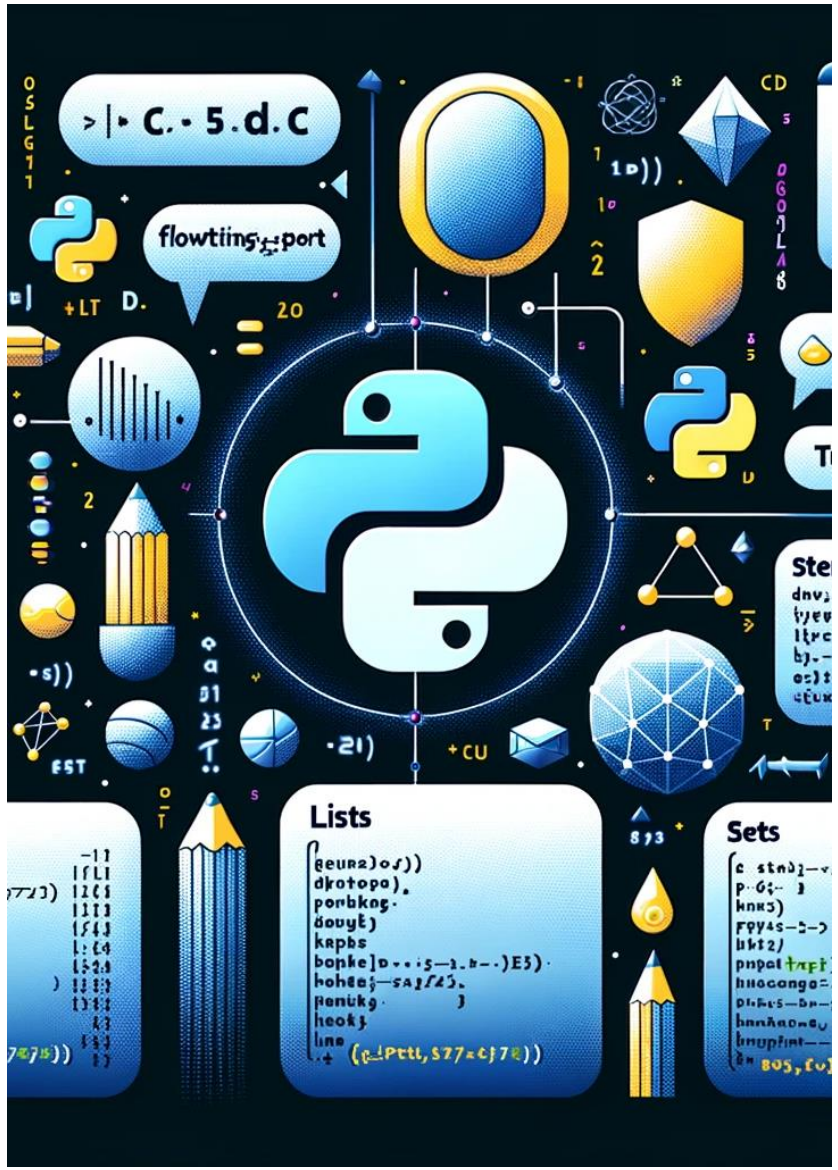
x *= 2

x /= 2

x //= 2

x %= 2

x **= 2



PYTHON DATA TYPES - STR

The str Type

str (String) is an ordered collection of characters, enclosed in single, double, or triple quotes. Examples:

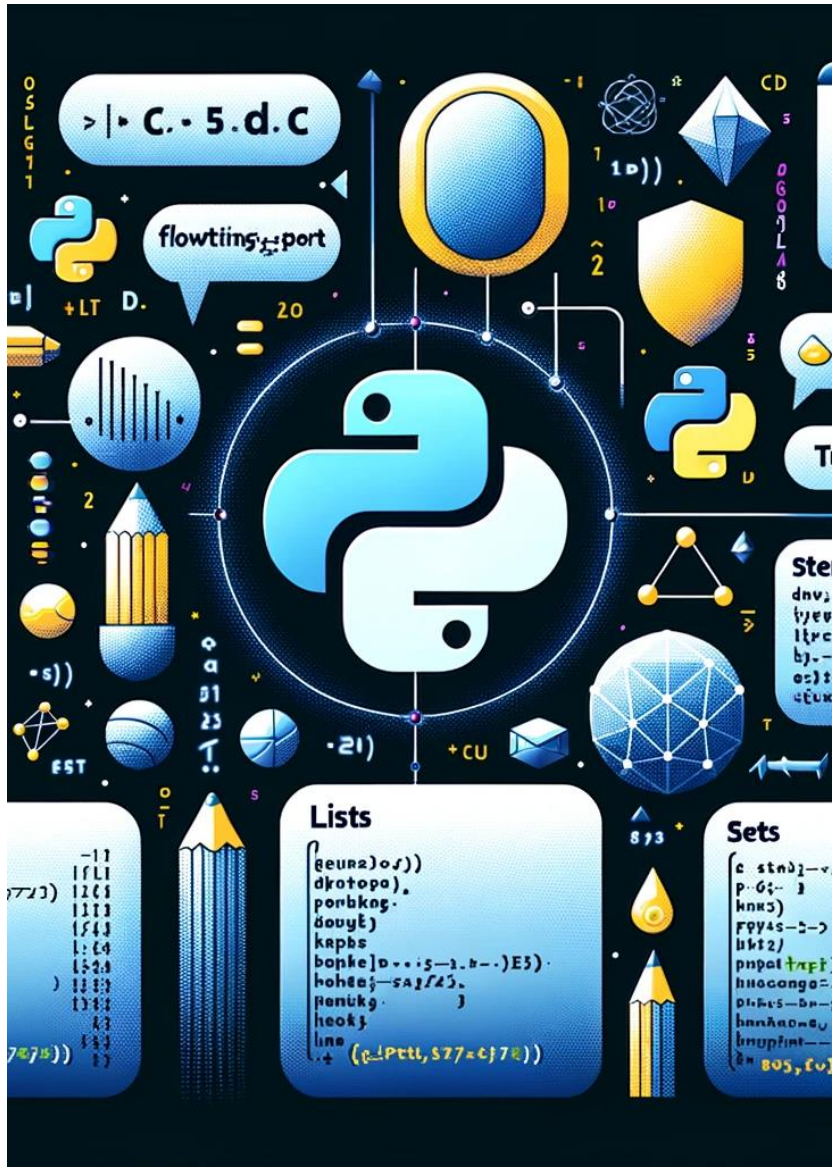
"Hello, World! "

‘when he saw me he said: "hi darling", it was cute'

"""multiline example

In python

Third line"""

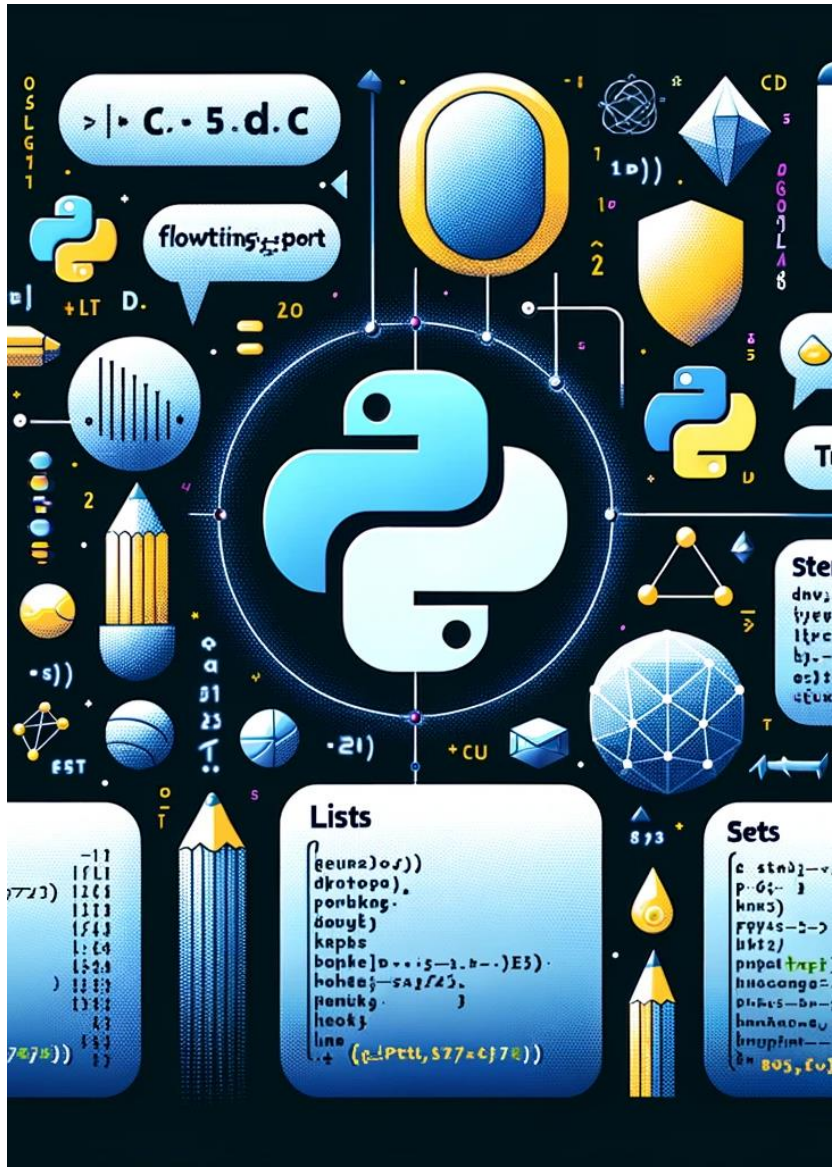


PYTHON DATA TYPES - STR

The str Type

Python's escape characters, are used within string literals to represent special characters or to invoke special actions

	Description	Example
\\	Backslash ()	"This is a backslash: \\\\"
'\	Single quote (')	'It\'s easy to use single quotes.'
"\	Double quote (")	"He said, \"Hello, World!\""
\n	New line	"First line.\nSecond line."
\t	Horizontal tab	"Column 1\tColumn 2"
\b	Backspace	"abc\bdef" -> "abdef"



PYTHON DATA TYPES - STR

raw strings:

- Python can define raw strings, that blocks escape characters
- Putting the r character before single, double or triple quotes string will define them as raw-string
- Raw-strings are used to backslash escape characters

to avoid escape characters, use:

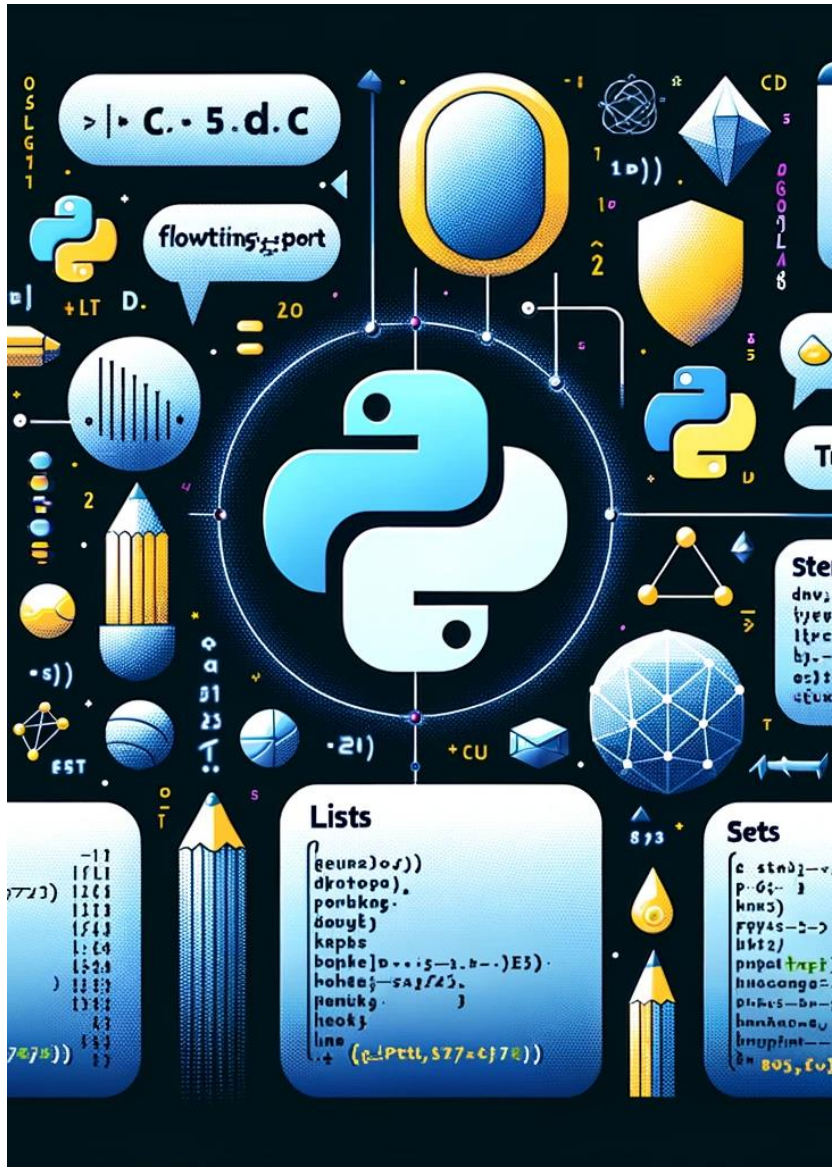
```
path = "C:\\temp\\newDir\\file.txt"
```

Or:

```
path = r"C:\temp\newDir\file.txt"
```

PYTHON DATA TYPES - STR

Operator		Description	Example
+	Concatenation	Joins two or more strings together	"Hello, " + "World!"
*	Repetition	Repeats a string a given number of times	"Python" * 3
[]	Indexing	Accesses a character at a specific position	"Python"[0] Output: 'P'
[:]	Slicing	Extracts a part of the string	"Hello, World!"[7:12] Output: 'World'
in	Membership	Checks if a substring exists within another string	"Py" in "Python" Output: True



PYTHON DATA TYPES - STR

String indexing

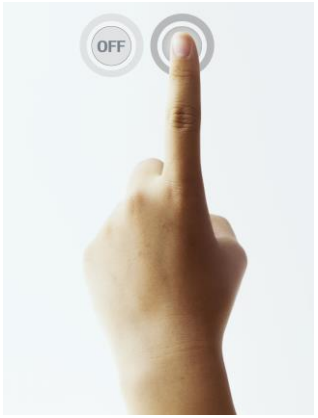
Strings can be indexed, with the first character having index 0 and last character having index -1

```
print('Python'[0])
```
output: **'P'**

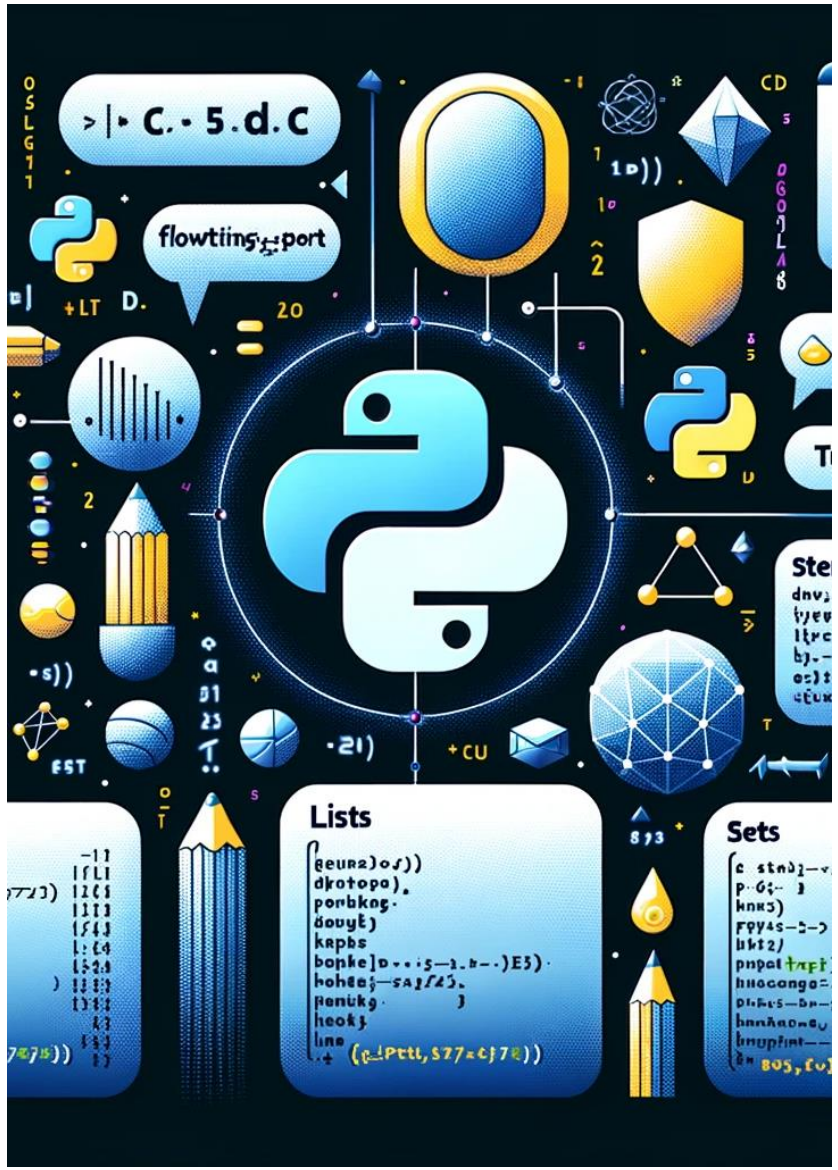
```
print('Python'[5])
```
output: **'n'**

```
print('Python’[-1)
```
output: **'n'**

```
print ('Python'[-2])
```
output: **'o'**



SQLabs
המכללה הגבוהה להייטק



PYTHON DATA TYPES - STR

String slicing

```
print('python'[0:2])
```

output: **py**

```
print('python'[2:5])
```

output: **tho**

```
print('python’[:2])
```

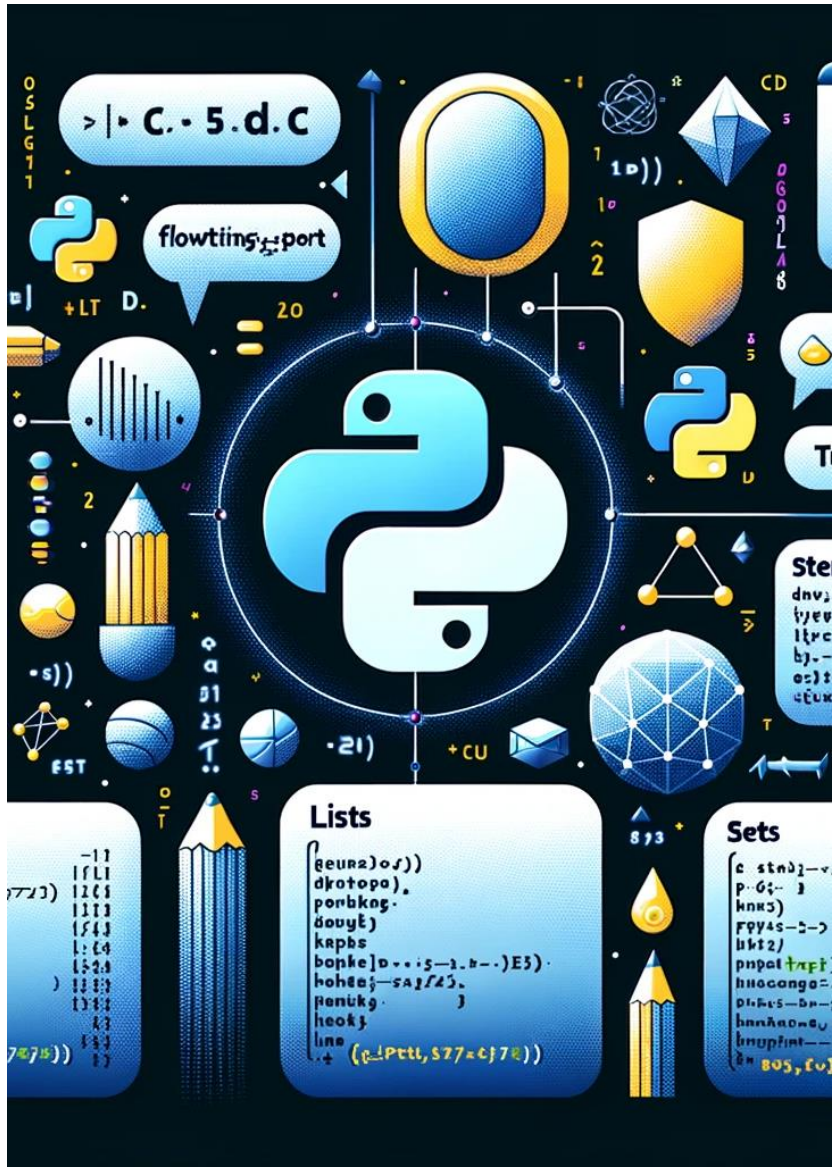
output: **py**

```
print ('Python’[4:])
```

output: **on**

```
print ('Python’[-2:])
```

output: **on**



PYTHON DATA TYPES - STR

String slicing

The third parameter in slicing is the step

[**start**:**end**:**step**]

```
print('python'[0::2])
```
output: **pto**

```
print('python'[::-1])
```
output: **nohtyp**

Python Data Types - str methods

Name	Description	Example
capitalize	Capitalizes the first letter of the string	"hello".capitalize() -> 'Hello'
upper	Converts all characters in the string to uppercase	"hello".upper() -> 'HELLO'
lower	Converts all characters in the string to lowercase	"HELLO".lower() -> 'hello'
strip	Removes leading and trailing whitespaces	" hello ".strip() -> 'hello'
find	Searches the string for a specified value and returns the position of where it was found	"hello".find("lo") -> 3
replace	Replaces a string with another string	"hello".replace("l", "r") -> 'herro'
count	Counts how many times a substring appears in the string	"hello".count("l") -> 2
startswith	Checks if the string starts with the specified substring	"hello".startswith("he") -> True
endswith	Checks if the string ends with the specified substring	"hello".endswith("lo") -> True
isdigit	Checks if all characters in the string are digits	"123".isdigit() -> True
isalpha	Checks if all characters in the string are alphabetic	"hello".isalpha() -> True

PYTHON DATA TYPES - LIST

The list type

List is an ordered and changeable collection of objects in any type. list is written as a comma-separated values between square brackets.

```
cubes = [1, 8, 27, 65, 125]
```

Like strings, lists can be indexed and sliced

```
cubes[3] = 64
print(cubes)
```

Python Data Types - list methods

Name	Description	Example
append	Adds an item to the end of the list	myList.append(5) -> Adds 5 to myList
extend	Adds all elements of a list to another list	myList.extend([8, 9]) -> Extends myList with [8, 9]
insert	Inserts an item at a given position	myList.insert(1, 'a') -> Inserts 'a' at index 1
remove	Removes the first item from the list that has a specified value	myList.remove('a') -> Removes 'a' from myList
pop	Removes the item at the given position (default is last)	myList.pop() -> Removes the last item from myList
clear	Removes all items from the list	myList.clear() -> Clears myList
index	Returns the index of the first item with a specified value	myList.index('a') -> Returns index of 'a'
count	Returns the number of times a specified value occurs	myList.count('a') -> Counts how many times 'a' appears
sort	Sorts the list	myList.sort() -> Sorts myList
reverse	Reverses the order of the list	myList.reverse() -> Reverses myList

LISTS

Analyze the data of the weather in a given week:
temperatures = [22.1, 23.4, 19.9, 25.4, 21.0, 20.3, 24.2]
Tasks:

1) Find the Highest Temperature.
2) Calculate the Average Temperature
3) Identify Days Above Average Temperature: Create a list of days (1-7, where 0 is the first day of the week) where the temperature was above the weekly average.

LISTS

Task 1: Find the Highest Temperature

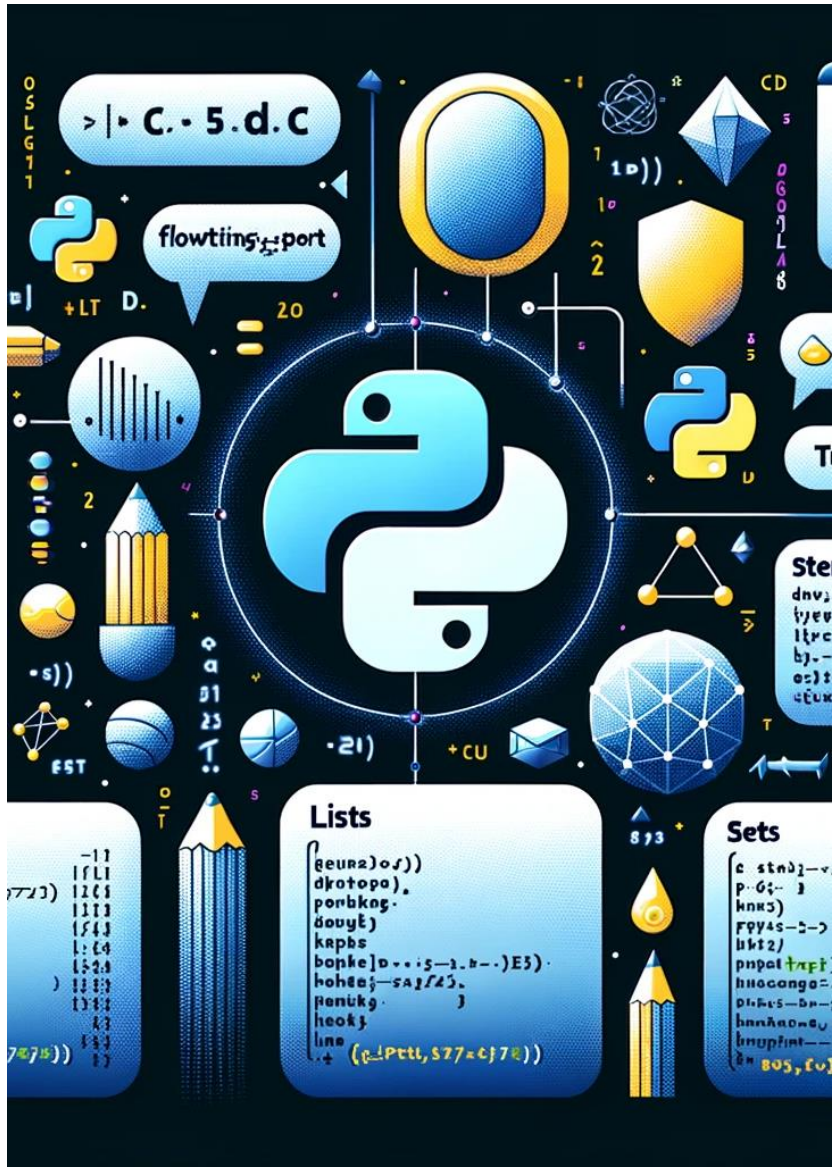
```
highest_temperature = max(temperatures)
print(f"Highest Temperature: {highest_temperature}°C")
```

Task 2: Calculate the Average

```
average_temperature = sum(temperatures) /
len(temperatures)
print(f"Average Temperature: {average_temperature:.2f}°C")
```

Task 3: Identify Days Above Average

```
days_above_average[] =
for i in temperatures:
    if i > average_temperature:
        days_above_average.append(temperatures.index(i)+(1
print(f"Days Above Average Temperature:
{days_above_average}")
```



PYTHON DATA TYPES - TUPLE

The tuple type

A tuple is a collection which is ordered and unchangeable.
Tuples are written with (or without) round brackets

```
my_data = ("hi", "hello", "bye")
print(my_data)
```

```
print(my_data[0])
```

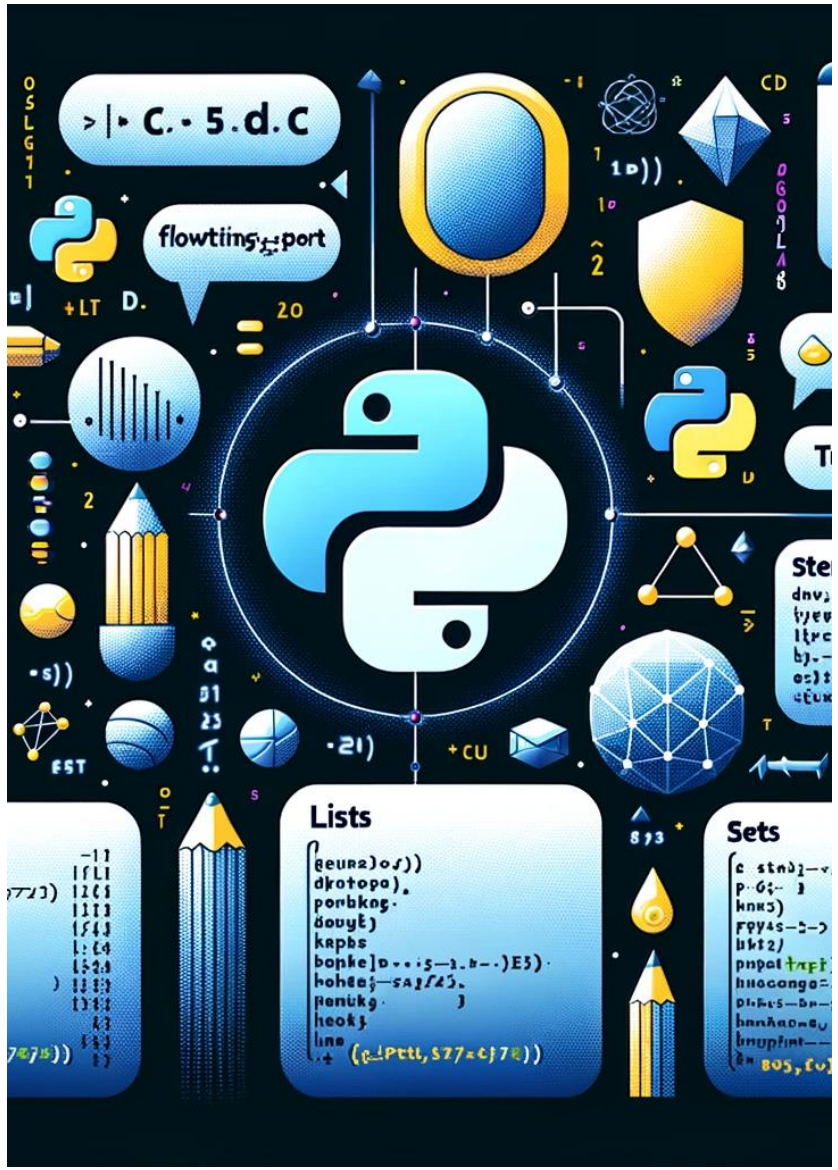
Output: hi

```
print(my_data[-1])
```

Output: bye

```
print(my_data[0:2])
```

Output: ('hi', 'hello')



PYTHON DATA TYPES - TUPLE

The tuple type

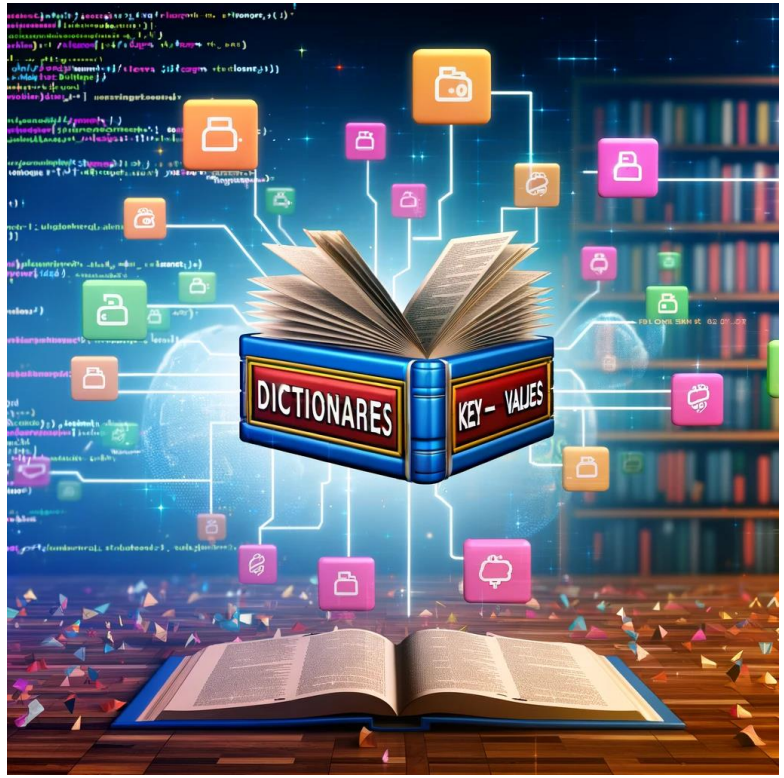
Choosing to use a tuple instead of a list in Python can be beneficial for several reasons:

Immutability: This is useful when you want to ensure that data remains constant and cannot be changed accidentally throughout your program.

Performance: Due to their immutability, tuples can be slightly faster than lists when it comes to iteration

Memory Efficiency: Since Python knows the tuple is immutable, it can allocate memory more efficiently.

PYTHON DATA TYPES - DICT



dict

a dictionary is a collection of items, within a curly braces {}.
every item is made up of two parts: a key and a value

```
my_dict = {'name': 'Alex', 'age': 12, 'favorite_color': 'Blue'}
```

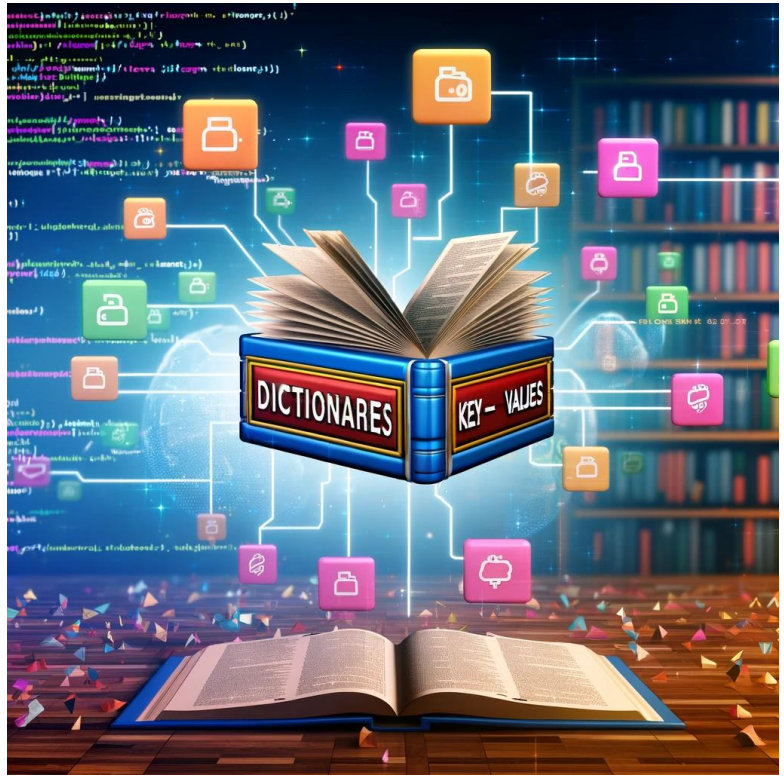
Get item:

```
print(my_dict['age'])
```

Set item:

```
my_dict['age'] = 13
```

PYTHON DATA TYPES - DICT



dict

Looping Through: You can loop through a dictionary to access all its keys, values, or both. For example, to print all the keys:

```
for key in my_dict.keys():
    print(key)
```

Python Data Types - dict methods

Name	Description	Example
clear()	Removes all items from the dictionary.	my_dict.clear()
get()	Returns the value for a key if it exists, otherwise returns None (or a specified default).	value = my_dict.get('key', 'default')
items()	Returns a view object containing key-value pairs of the dictionary.	for key, value in my_dict.items(): print(key, value)
keys()	Returns a view object containing the keys of the dictionary.	for key in my_dict.keys(): print(key)
pop()	Removes the item with the specified key and returns its value.	value = my_dict.pop('key')
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value.	value = my_dict.setdefault('key', 'default')
update()	Updates the dictionary with the specified key-value pairs.	my_dict.update({'key': 'value'})
values()	Returns a view object containing the values of the dictionary.	for value in my_dict.values(): print(value)

DATA TYPES - DICT

Write a Python program to count the character frequency in a string.
Sample String : ' google.com’

Expected Result :
{'g': 2, 'o': 3, 'l': 1, 'e': 1, '.': 1, 'c': 1, 'm': 1}

DATA TYPES - DICT

```
word = "google.com"
dict1 = {}
for char in word:
    dict1[char] = dict1.setdefault(char,0)+1

print(dict1)
```

unpacking

When you have a list or tuple, you can unpack its elements into individual variables.

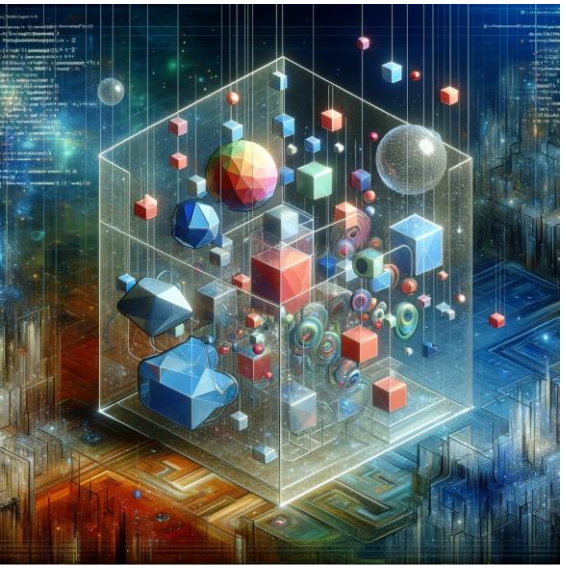
```
numbers = (1, 2, 3)
a, b, c = numbers
print(a)   # Output: 1
print(b)   # Output: 2
print(c)   # Output: 3
```

unpacking

Python 3 introduced an extended unpacking feature that uses an asterisk * to grab excess items.

```
numbers = [1, 2, 3, 4, 5]
first, *middle, last = numbers
print(first)        # Output:1
print(middle)   # Output:[2,3,4]
print(last)         # Output:5
```

PYTHON DATA TYPES - SET



A "set" in Python is a useful collection that can make certain tasks in programming much easier and more efficient.

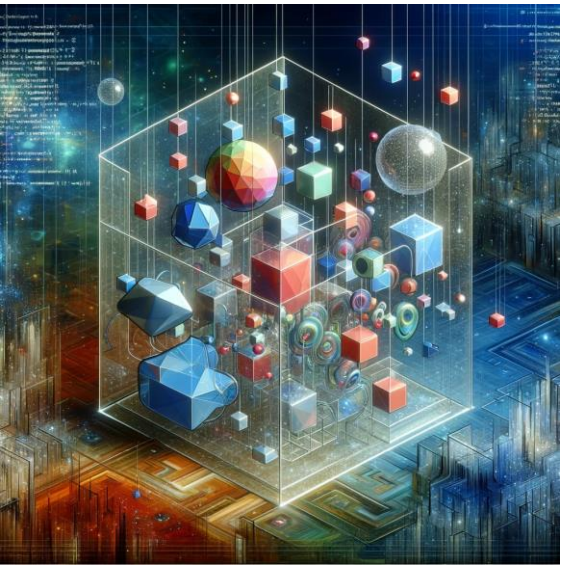
Uniqueness: The most important feature of a set is that it does not allow duplicates. This is useful when you need to ensure all items are unique or when you want to remove duplicates from a collection.

Creating a Set: You can create a set using curly braces {} with items separated by commas, or by using the set() function

```
my_fruits = {'apple', 'banana', 'cherry'}
```

```
my_list = ['apple', 'banana', 'cherry', 'apple', 'cherry']
my_fruits = set(my_list)  # This will remove duplicates
```

PYTHON DATA TYPES - SET



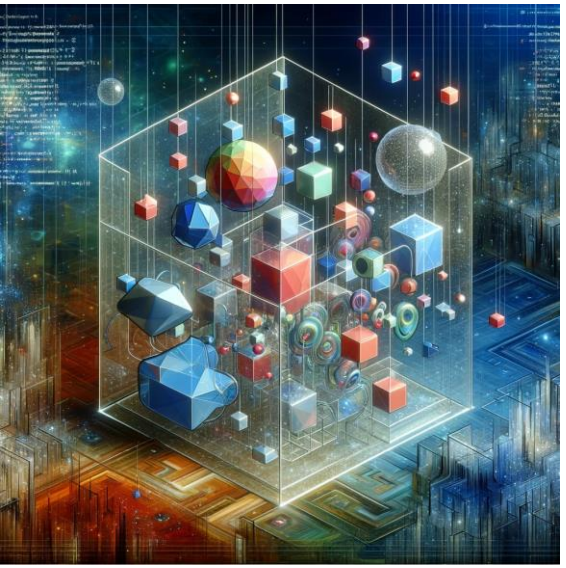
Adding Items: You can add items to a set using the .add() method. For example:

```
my_fruits.add('orange')
```

Removing Items: To remove an item, you can use the .remove() or .discard() methods. The difference is that if the item doesn't exist, .remove() will raise an error, while .discard() will not do anything, which can be quite handy to avoid errors.

```
my_fruits.remove('banana')  # Removes 'banana', or raises an error if not found
my_fruits.discard('pear')  # Removes 'pear' if it exists, does nothing if not
```

PYTHON DATA TYPES - SET



Operations: Sets support mathematical set operations like union, intersection, difference, and symmetric difference. These can be useful for comparing sets:

Union (|): Combines the elements of two sets.
Intersection (&): Gets items only in both.
Difference (-): Gets items in the first set but not in the second.
Symmetric Difference (^): Gets items in either set, but not in both.:

```
a = set('abracadabra')
b = set('abc')
print(a & b) # letters in both a and b
print(a.intersection(b))
print(a ^ b) # letters in a or b but not both (XOR)
```

SQLabs
המכללה הגבוהה להייטק

Python Data Types - set methods

Name	Description	Example
add()	Adds an element to the set.	my_set.add('apple')
clear()	Removes all elements from the set.	my_set.clear()
copy()	Returns a shallow copy of the set.	new_set = my_set.copy()
difference()	Returns a set containing the difference between two or more sets.	new_set = set1.difference(set2)
discard()	Removes an element from the set if it is a member. If the element is not a member, does nothing.	my_set.discard('apple')
intersection()	Returns a set that is the intersection of two or more sets.	new_set = set1.intersection(set2)
isdisjoint()	Returns True if two sets have a null intersection.	result = set1.isdisjoint(set2)
issubset()	Returns True if another set contains this set.	result = set1.issubset(set2)
pop()	Removes and returns an arbitrary set element. Raises KeyError if the set is empty.	element = my_set.pop()
remove()	Removes an element from the set. If the element is not a member, raises a KeyError.	my_set.remove('apple')
union()	Returns a set that is the union of two sets.	new_set = set1.union(set2)

CONDITIONAL STATEMENTS – IF ELIF ELSE

The most commonly used conditional statements are **if**, **elif** (else if), and **else**.

```
if age < 18:
    print("You are not eligible to vote.")
elif age >= 18:
    print("You are eligible to vote.")
```

Ternary Operators, or **Conditional Expressions**

```
a = 8
b = 12
print("A") if a > b else print("B")
```

SQLabs
המכללה הגבוהה להייטק

CONDITIONAL STATEMENTS – WALRUS OPERATOR

the walrus operator were added to Python in version 3.8

Instead of:

```
str = "hello python"
if len(str) > 10:
    print(f'string is too long {len(str)} characters')
```

do

```
str = "hello python"
if (x := len(str)) > 10:
    print(f'string is too long {x} characters')
```

CONDITIONAL STATEMENTS – match case

Match-case operators were added to Python in version 3.10

```
day = 5
match day:
    case day if day < 5:
        print('working day')
    case 6:
        print('Friday')
    case 7:
        print('Saturday')
    case _:
        print('Thursday')
```

LOOPS

Loops allow you to execute a block of code repeatedly, as long as a condition is met.
Python provides two types of loops: **for** loops and **while** loops

for Loop: Iterates over a sequence (like a list, tuple, dictionary, set, or string) and executes a block of code for each item .

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

while Loop: Repeatedly executes a block of code as long as a condition is True.

```
count = 0
while count < 5:
    print(count)
    count += 1
```

The Range object

The range object generates a sequence of numbers, but unlike a list, it doesn't store all those numbers simultaneously in memory. Instead, it calculates each number on the fly as needed, which makes it more memory efficient, especially for large ranges.

the range() function can take one, two, or three arguments:
Examples
range(5) generates numbers from 0 to 4.
range(1, 5) generates numbers from 1 to 4.
range(0, 10, 2) generates even numbers from 0 to 8.

CONTROL STATEMENTS

Control statements alter the execution flow of loops and conditional blocks. The three main control statements are **break**, **continue**, and **pass**.

break: Exits the loop in which it is placed. It is typically used to exit a loop when a certain condition outside the normal loop condition is met.

```
for number in range(1, 10):
    if number == 5:
        break  # Exit the loop when number is 5
    print(number)
```

continue: Skips the rest of the code inside the loop for the current iteration and moves to the next iteration of the loop.

```
for number in range(1, 10):
    if number == 5:
        continue  # Skip 5
    print(number)
```

A prime number is a number greater than 1 that has no positive divisors other than 1 and itself.

Write a Python script to decide if a given number is primary or not.


```
num = int(input("enter a number"))
if num < 2:
    print("not a primary number")
for i in range(2, int(num**0.5) + 1):
    if num % i == 0:
        print("not a primary number")
        break
else:
    print(" a primary number")
```

The else part of a for loop executes only when the loop completes normally (without a break).

FUNCTIONS

```
# basic function - no return value
def print_hello():
    print("Hello")
```

```
print_hello()
```

```
# basic function - returns value
def return_hello():
    return "Hello"
```

```
print(return_hello())
```

```
# function with parameter
def greet_user(user_name):
    print(f"Hello {user_name}, Nice to meet you")
```

```
greet_user("John")
```

SQLabs
המכללה הגבוהה להייטק

Write a Python function that checks whether a passed string is palindrome or not (A palindrome is a word that reads the same backward as forward, e.g., madam)

```
def is_palinderome(str):
    return str == str[::-1]
```

FUNCTIONS-DEFAULT VALUE

```
def greeting_with_default(first_name, time_of_day="morning"):
    s = f"Hello {first_name} good {time_of_day}"
    return s

print(greeting_with_default("John"))
```

Output:
Hello John good morning

FUNCTIONS-KEYWORD ARGUMENTS

```
def greeting(first_name, time_of_day):
    s = f"Hello {first_name} good {time_of_day}"
    return s


# ordered arguments
print(greeting("David", "morning"))
# Keyword arguments
print(greeting(time_of_day="morning",first_name="David"))
```

FUNCTIONS arbitrary argument lists (*args)

```
def add(*args):
    total = 0
    for i in range(len(args)):
        total += args[i]
    return total
```

```
print(add(6,7,8,6,5,4,3))
```

Output: 39

Write a Python function that accepts an unknown numbers of arguments and returns True if the arguments form a series of ascending numbers

```
def is_ascending(*numbers):
    for i in range(len(numbers) - 1):
        if numbers[i] >= numbers[i + 1]:
            return False
    return True
```


```python
def is_ascending(*numbers):
    return list(numbers) == sorted(numbers)
```

FUNCTIONS

The ****kwargs** syntax in Python is used in function definitions to handle named arguments that you have not explicitly defined beforehand. The ** is what tells Python to collect all remaining keyword arguments into a dictionary with the argument names as the keys and their corresponding values as the dictionary values.

```
def greet(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")


greet(first_name="John", last_name="Doe")
```

```
C:\python_projectt\venv\Scripts\python.exe
first_name: John
last_name: Doe
```

FUNCTIONS

Combining *args and **kwargs:
You can use *args and **kwargs together in functions to allow for a flexible number of positional and named arguments:

```
def example_func(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)


example_func( *args: 1, 2, 3, first_name="John", last_name="Doe")
```

```
C:\python_projectt\venv\Scripts\python.exe C:\python_projectt\o
Positional arguments: (1, 2, 3)
Keyword arguments: {'first_name': 'John', 'last_name': 'Doe'}
```

SQLabs
המכללה הגבוהה להייטק

LOCAL AND GLOBAL VARIABLES

- The scope of a variable determines where it can be accessed.

- Variables that are defined inside a function body have a local scope. Local variables can be accessed only inside the function in which they are declared

- Variables that are defined outside any function have a global scope. Global variables can be accessed throughout the program body by all functions.

LOCAL AND GLOBAL VARIABLES

```
def func():
    print("in func: ", glob)


glob = 10
func()
print("in main space: ", glob)
```

```
C:\python_projectt\venv\Scripts\python.exe
in func:  10
in main space:  10
```

LOCAL AND GLOBAL VARIABLES

```
glob = 10
def func():
    glob = 11
    print("in func: ", glob)

func()
print("in main space: ", glob)
```

```
C:\python_projectt\venv\Scripts\python.exe
in func:  11
in main space:  10
```

LOCAL AND GLOBAL VARIABLES

```python
glob = 10
def func():
    global glob
    print("in func: ", glob)
    glob = 11
    print("in func: ", glob)

func()
print("in main space: ", glob)
```

```
C:\python_projectt\venv\Scripts\python.exe
in func:  10
in func:  11
in main space:  11
```

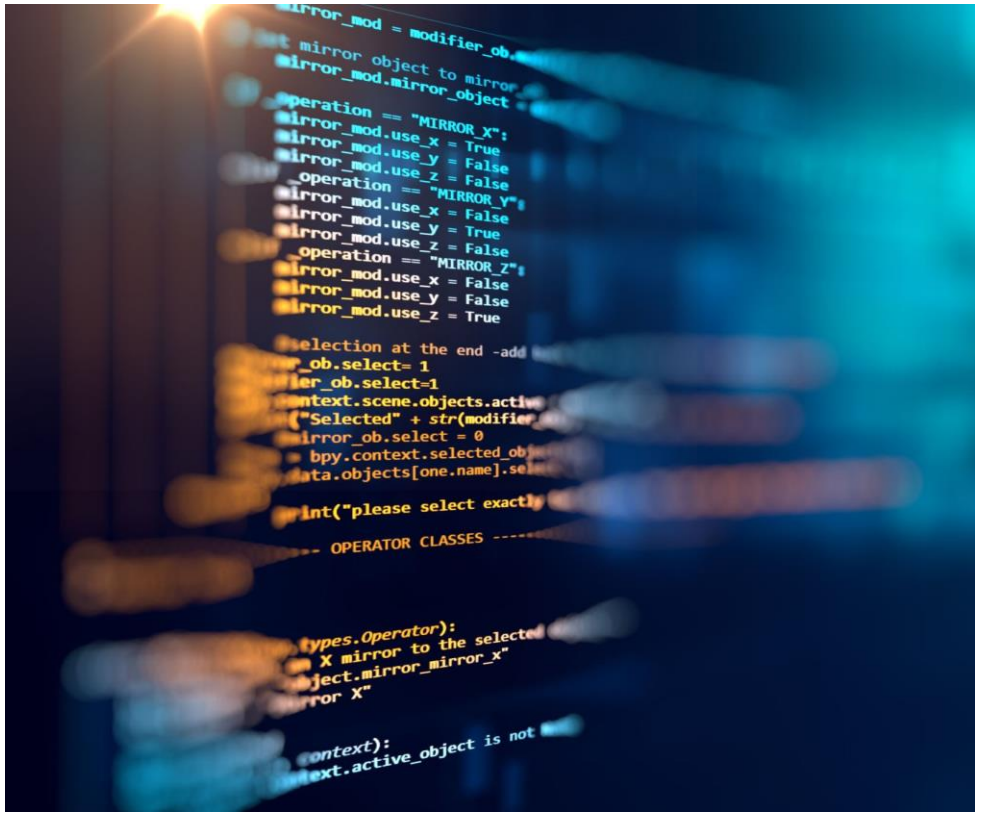
PASS BY REFERENCE VS VALUE

- mutable parameters in the Python language are passed by reference.
- immutable parameters in the Python language are passed by value

```
def chang_list( lst ):
    "This changes a passed list into this function"
    lst.append(4)
    print ("inside the function: ", lst)

list = [1,2,3]
chang_list(list )
print ("outside the function: ", list)
```

Modules and packages



- Creating and Importing Modules
- Using Built-in Module
- Packages and Package Management (pip)

SQLabs
המכללה הגבוהה להייטק

What is a Module?

A module in Python is simply a file containing Python code. It may define functions, classes, and variables, and can also include runnable code. Grouping related code into a module makes the code easier to understand and use.

To create a module, you simply need to save your code in a .py file.

Importing a Module

Once you have created a module, you can use it in other Python scripts by importing it using the import statement.

```
import mymodule
```

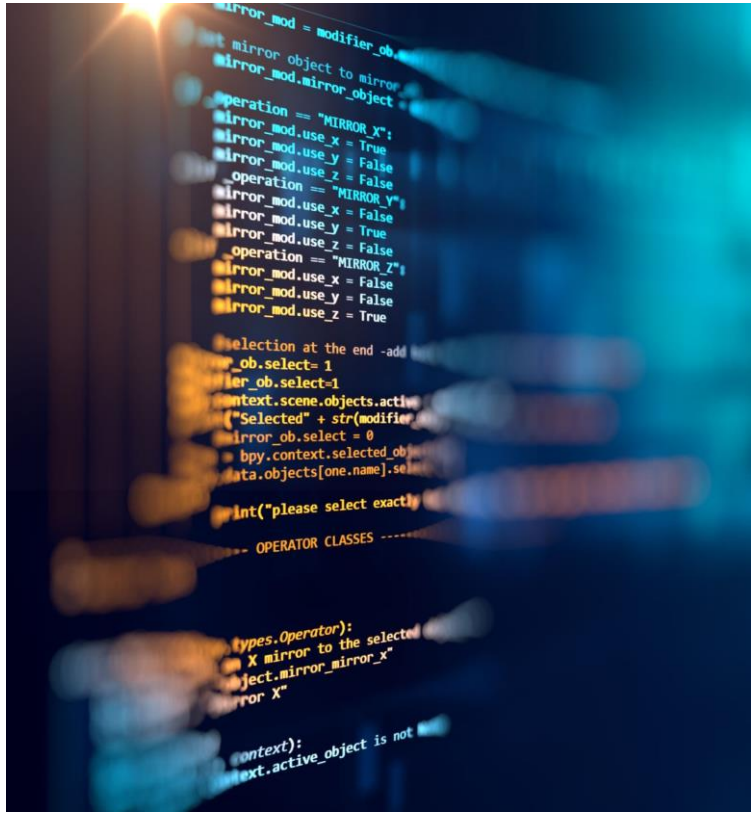
Importing Specific Attributes: You can also choose to import specific attributes (functions, classes, variables) from a module instead of the entire module:

```
from mymodule import greet, Calculator
```

ributes upon importing (for instance, to avoid naming conflicts or for convenience), you can use the as keyword:

```
import mymodule as mm
from mymodule import Calculator as Calc
```

Object Oriented Programming



- Creating a class
  - Class built in methods (constructor, destructor, operands, etc.)
- Type of methods: instance, static, class method
  - Class instances, instance methods
- Inheritance and Polymorphism
  - Abstract method and abstract class

SQLabs
המכללה הגבוהה להייטק

Classes and objects

Object-Oriented Programming (OOP) in Python revolves around creating classes and objects.

It's like building a blueprint for a house (class) and then constructing houses (objects) based on that blueprint. Each house can have its own unique characteristics (attributes) and things it can do (methods).

In Python, classes can have three types of methods: instance methods, static methods, and class methods.

Each serves a different purpose and interacts with the class and its objects in unique ways.

Classes and objects

Python defines a set of predefined types of objects, like int, string, list,
User can define its own, user defined type of object using *class* keyword

```
class Point():
    def show(self):
        print("i am a point")


p1 = Point()
p1.show()
```

The first parameter of methods is the instance the method is called on

This parameter usually called **self**

built in methods

classes may define special methods, with predefined names and meaning and format like __XXX__.
They usually used for operators overloading and built-ins overriding
They are automatically invoked
For example:
__init__ - responsible for class instantiation for the newly-created class instance
__str__ - returns string representation of an object
Methods for arithmetic operations:
 __add__ addition
__gt__ greater than
__ge__ greater or equal to

builtin methods

```
class Point():
    def __init__(self,x=0,y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return f"A Point ({self.x},{self.y})"

    def __gt__(self, other):
        return self.x**2+self.y**2>other.x**2+other.y**2
```

```
p1 = Point(5,3)
p2 = Point(4,4)
print(p1>p2)
```

SQLabs
המכללה הגבוהה להייטק

Static Methods

Static methods are methods that don't access or modify the class state or the instance's state.

They are utility functions that perform a task in isolation.

You can think of them as functions that belong to a class but don't need a reference to any object or class.

To define a static method, use the @staticmethod decorator.

Static Methods

```
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y
```

Static methods can be called using the class name or an instance of the class, but they don't require an instance to be called.

```
print(MathOperations.add(5, 7))  # 12
```

Class Methods

Class methods are methods that operate on the class itself and not on instances of the class.
They can modify the class state that applies across all instances of the class.
The first parameter of a class method is a reference to the class, conventionally named cls.
Class methods are defined with the @classmethod decorator.

Class Methods

Class methods can be called using the class name, and they are used often for factory methods that return instances of the class.

```
class Dog:
    _dogs_count = 0

    def __init__(self, name):
        self.name = name
        Dog._dogs_count += 1

    @classmethod
    def dogs_count(cls):
        return cls._dogs_count
```

```
Dog('Buddy')
Dog('Lucy')
print(Dog.dogs_count())  # 2
```

Methods Summary

Instance Methods: Operate on an instance of a class. Use **self** to access the instance.

Static Methods: Don't operate on the class or instance. Use @staticmethod decorator. They are like functions that belong to a class's namespace.

Class Methods: Operate on the class itself. Use **cls** to access the class. They can modify class state and are defined with @classmethod decorator.

Class Inheritance

Inheritance and polymorphism are fundamental concepts of object-oriented programming (OOP) that allow for the creation of a more organized and modular code structure.

Inheritance allows a class (known as a child or subclass) to inherit attributes and methods from another class (known as a parent or superclass). This means you can create a new class based on an existing class, but with some additions or changes. It's like getting an old recipe from your grandparent and tweaking it to add your own flavor.

Class Inheritance

In inheritance, the class that performs the inheritance called **derived class** and the one who we inherits (extends) from - called **base class**

The **child class** inherits all attributes of its parent class
A **derived class** can override any method of its base class, and a method can call the method of a base class with the same name

```
class derivedClass (BaseClass1 [, BaseClass2, ...]):
    'Optional class documentation string'
    commands
```

Class Inheritance

```python
class ColoredPoint(Point):
    def __init__(self,x,y,color="White"):
        super().__init__(x,y)
        self.color = color

    def __str__(self):
        return f"{super().__str__()}, my color is {self.color}"
```

```python
p1 = ColoredPoint(4,3,"green")
print(p1)
```

```
C:\Users\david\PycharmProjects\sony\.venv\Scripts\python.exe
A Point (4,3), my color is green
```

SQLabs
המכללה הגבוהה להייטק

Method Resolution Order

MRO stands for Method Resolution Order.
It's a rule that Python follows to determine the order in which classes are searched when you're working with inheritance, especially in complex scenarios involving multiple inheritance.
This concept is crucial because it defines how Python will look for methods and attributes across different base classes.

In the context of inheritance, Python allows a class to inherit from multiple parent classes, a feature known as multiple inheritance.

However, multiple inheritance can introduce ambiguities when the same method or attribute is defined in more than one parent class. To resolve these ambiguities, Python uses a specific order called MRO.

Method Resolution Order

Python uses the C3 linearization algorithm for computing the MRO in classes. This algorithm ensures that:

1. A class always appears before its parents.
2. The order of parent classes is preserved.

To see the MRO of a class, you can use the .__mro__ attribute or the mro() method on the class.

Method Resolution Order

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass
```

In this example, D inherits from both B and C. The MRO for class D determines the order in which Python searches for methods and attributes. According to the C3 linearization

What would be the MRO for D?

```
D, B, C, A, object
```

Polymorphism

Example with Polymorphism:

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

    def speak(self):
        pass

class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        return f"{self.name} says Woof!"

class Cat(Animal):  # Cat inherits from Animal
    def speak(self):
        return f"{self.name} says Meow!"
```

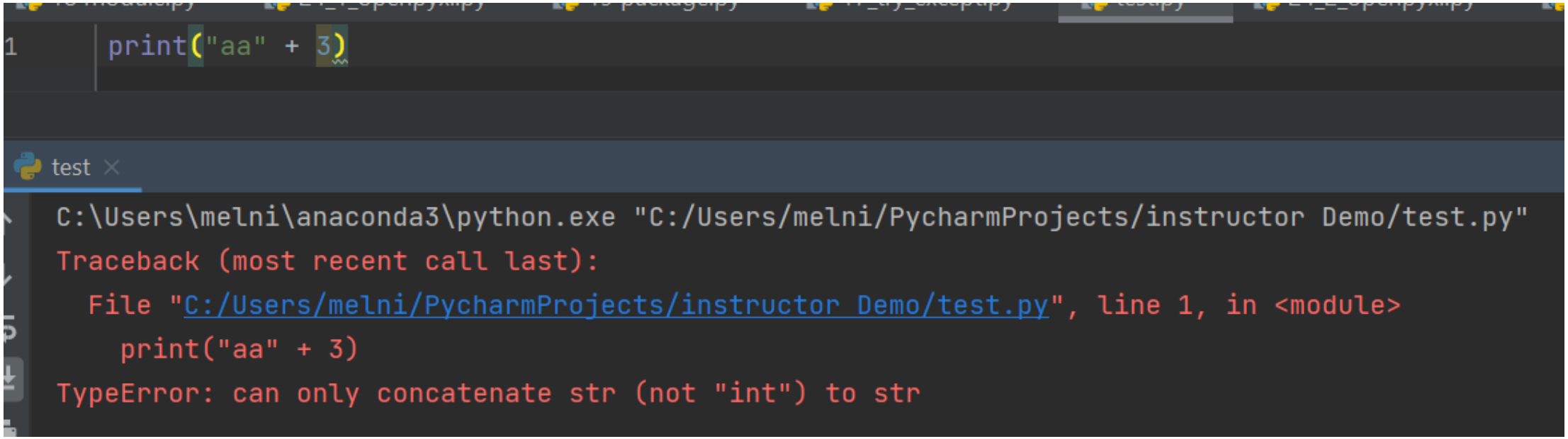
```python
def animal_speak(animal):
    print(animal.speak())

buddy = Dog("Buddy")
kitty = Cat("Kitty")

animal_speak(buddy)  # Buddy says Woof!
animal_speak(kitty)  # Kitty says Meow!
```

Errors & Exceptions

When an error occurs, Python will stop and generate an error message.



SQLabs
המכללה הגבוהה להייטק

Errors & Exceptions **try-except**

errors can be handled using the **try-except** statement
In the following example, The try block will generate an error, because x is not defined

```
try:
  print(x)
except:
  print("An exception occurred")
```

the except block will be executed.

Errors & Exceptions **try-except**

You can define as many exception blocks as you want.
you can execute a special block of code for a specific error

```
try:
  print(x/y)
except ZeroDivisionError:
  print("division in zero is not allowed")
except TypeError:
  print("variables must be on the same type")
except:
  print("unknown error")
```

Except with tuple of values

```
try:
    # do something
    pass
except ValueError:
    # handle ValueError exception
    pass
except (TypeError, ZeroDivisionError):
    # handle multiple exceptions
    pass
except:
    # handle all other exceptions
    pass
```

We can use a **tuple** of values to specify multiple exceptions in an except clause

try-except-else

Use the **else** keyword to define a block of code to be executed if no errors were raised:

Example: In this example, the try block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

try-except-finally

The **finally** block, if specified, will be executed regardless if the try block raises an error or not.

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

Raising Exceptions

manually raise exceptions using the **raise** keyword

```
try:
    a = int(input("Enter a positive integer: "))
    if a <= 0:
        raise ValueError("That is not a positive number!")
except ValueError as ve:
    print(ve)
```

Output:

Enter a positive integer: -1

That is not a positive number!

User defined Exceptions

User can define custom exceptions by creating a new class

This exception class has to be derived, either directly or indirectly, from the built-in Exception class

User defined Exceptions

```
class Error(Exception):
   pass

class TransitionError(Error):
   def __init__(self, prev, nex, msg):
      self.prev = prev
      self.next = nex

      self.msg = msg
try:
   raise(TransitionError(2,3*2,"Not Allowed"))

except TransitionError as error:
   print('Exception occured: ',error.msg)
```

SQLabs
המכללה הגבוהה להייטק

COMPREHENSIONS

- Comprehensions are efficient and readable tools for creating **lists** and **dictionaries** from existing **iterables**.
- They encapsulate looping and conditional logic in a single line, making your code cleaner and often faster.
- Practice using these features with different data types and structures to become proficient in their application and to fully leverage the elegance and power they bring to Python programming.

LIST COMPREHENSIONS

List comprehensions provide a concise way to create lists. The basic syntax is:

```
[expression for item in iterable if condition]
```

Example 1: Simple List Comprehension

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

print(squares)
```

Example 2: List Comprehension with Condition

```
even_numbers = [x for x in range(10) if x % 2 == 0]

print(even_numbers)
```

DICTIONARY COMPREHENSIONS

Dictionary comprehensions are similar to list comprehensions but for dictionaries. They allow you to construct dictionaries in a single, readable line. The basic syntax is:

```
{key_expression: value_expression for item in iterable if condition}
```

This allows for dynamically creating dictionary keys and values based on the iterating item and an optional condition.

Example 1: Simple Dictionary Comprehension

```
squares_dict = {x: x**2 for x in range(10)}
print(squares_dict)
```

Example 2: Dictionary Comprehension with Condition

```
even_squares_dict = {x: x**2 for x in range(10) if x % 2 == 0}
print(even_squares_dict)
```

COMPREHENSIONS

Best Practices

- **Readability**: Comprehensions should be used to enhance readability. If a comprehension gets too complex, consider using regular loops instead.
- **Performance**: Comprehensions can be faster than equivalent loops due to internal optimizations, especially for large datasets.
- **Conditionals**: Use conditionals sparingly in comprehensions to keep them understandable.
- **Nested Comprehensions**: Be cautious with nesting, as it can quickly lead to less readable code.

lambda function

```
x = lambda a : a + 10
print(x(5))
```

Output:
15

```
x = lambda a, b : a * b
print(x(5, 6))
```

Output:
30

lambda function inside a function

```
def myfunc(n):
  return lambda a : a * n
mydoubler = myfunc(2)
print(mydoubler(11))
```

Output:
22

```
mytripler = myfunc(3)
print(mytripler(11))
```

Output:
33

SQLabs
המכללה הגבוהה להייטק

filter function

resultIter = filter(function, iterable)

- filter function returns an iterator of elements of iterable for which function returns true. iterable may be either a sequence or an iterator
- The filter iterates through all elements of iterable, sends them(one by one) to function and includes in resultIter only elements for which the function returns True
- If the function is not None, elements from input iterable will be include in a result only if them evaluates as True

filter function

For example:

```
list= [2, 18, 9, 22, 17, 24, 8, 12, 27]
seq = filter(lambda x: x % 3 == 0, list)

for val in seq:
        print(val)              # 18, 9, 24, 12, 27

seq = filter(lambda x: x < 0, range(-5,5))
for val in seq:
        print(val)              # -5, -4, -3, -2, -1
```

map function

resultIter = **map**(function, iterable, [iterables])

Return an iterator that applies function to every item of iterable.

If additional iterables arguments are passed, function must take that many arguments and is applied to the items from all iterables corresponding

If the number of items in all iterables is not even, the map function will only iterate through the common items, in python 3 (the number of iterations is defined by minimal sequence length)

MAP FUNCTION

For Example:

```
list = [2, 18, 9, 22, 17, 24, 8, 12, 27]
map(lambda x: x * 2 + 10, list)   # 14, 46, 28, 54, 44, 58, 26, 34, 64

map(lambda w: len(w), 'A lot of cats and dogs here'.split())
                                        #1, 3, 2, 4, 3, 4, 4

map(pow, [2, 3, 4], [10, 5, 1, 3])    # 1024, 243, 4
```

iterators

Iterators in Python are objects that allow you to iterate over a collection, such as a list or a tuple.

They follow the iterator protocol, which means they support two fundamental methods: __iter__() and __next__().

Understanding iterators is key to working efficiently with loops and generating sequences of values without needing to store the entire sequence in memory at once.

iterators

The Iterator Protocol

- **__iter__()** Method: This method returns the iterator object itself. It is called when iteration is initialized, for example, by the **iter()** function or loops.

- **__next__()** Method: This method returns the next item from the collection. If there are no more items, it raises the **StopIteration** exception, signaling that the iteration is complete.

iterators

You can make any object an iterator by implementing the iterator protocol

```
class Count:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1
```

In this example, Count is an iterator that generates numbers from low to high. It can be used in a loop like this:

```
for number in Count(1, 3):
    print(number)
```

This will print:

```
1
2
3
```

SQLabs
המכללה הגבוהה להייטק

iterators

Using Iterators

Python's built-in containers (like **lists**, **tuples**, **dictionaries**) are iterable, meaning they return an **iterator** when passed to the **iter()** function. You can manually iterate over these containers like so:

```
my_list = [1, 2, 3]
my_iter = iter(my_list)

print(next(my_iter))  # Output: 1
print(next(my_iter))  # Output: 2
print(next(my_iter))  # Output: 3
```

When **next**(my_iter) is called after the last item, a **StopIteration** exception is raised, indicating that the iteration is finished.

iterators

Why Use Iterators?

- **Memory Efficiency**: Iterators don’t require all the items to be stored in memory at once, which is especially useful for large datasets.

- **Laziness**: Iterators compute one item at a time, only when you ask for it, which can lead to performance gains.

- **Universality**: The iterator protocol provides a standard way to loop through any iterable object in Python, making code more Pythonic and reducing the need for index-based loops.

In summary, iterators are a fundamental part of Python that provide a flexible and efficient way to iterate over data. By understanding and using iterators, you can write cleaner, more efficient, and more Pythonic code.

generators

Generators in Python are a special type of iterator – a simple and powerful tool for creating iterators.

They are written like regular functions but use the **yield** statement to return data.

Every time a generator's next() method is called, it resumes execution right after the yield statement where it left off in the previous call.

This behavior makes generators a very efficient way to handle sequences of data without needing to store them all in memory at once.

generators

How Generators Work

When you call a generator function, it doesn't run the code immediately. Instead, it returns a generator object. Execution starts when next() is called on this object. Here's a basic example:

```
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()  # This doesn't execute my_generator()

print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

After yielding all its values, further calls to next() will raise a StopIteration exception, signaling that the iteration is complete.

generators

Advantages of Generators

• **Memory Efficiency**: Generators facilitate the creation of iterators with minimal memory usage because they yield items one at a time and only when required.

•**Convenience**: Writing a generator is often more concise and readable than building an iterator using a class.

•**Composition**: Generators can be composed together, meaning you can use a generator to process another generator's output. This allows for efficient data pipelines and transformations.

generators

Generator Expressions

Python also supports generator expressions, a high-level syntax that allows creating generators in a way similar to list comprehensions. For example:

```
gen_exp = (x**2 for x in range(4))  # This is a generator expression


print(next(gen_exp))  # Output: 0
print(next(gen_exp))  # Output: 1
print(next(gen_exp))  # Output: 4
```

Generator expressions are more memory-efficient than list comprehensions for large datasets, as they produce items one at a time, on demand.

SQLabs
המכללה הגבוהה להייטק

generators

Generators are particularly useful when:

•**Dealing with Large Datasets**: They allow you to process large datasets without the need to load everything into memory.

•**Representing Infinite Sequences**: Generators can represent an infinite sequence of data without any issues, as they only produce one item at a time.

•**Pipelining Data Transformations**: Generators can be used to pipeline a series of transformations, making your code more readable and efficient.

Summary: generators are a powerful feature in Python that enable you to write efficient and memory-friendly code for iterating over sequences. They are especially useful for processing large datasets and creating complex data pipelines with minimal overhead.

Closures

- Dynamically created functions that are returned by other functions.
- Full access to the variables and names defined in the local namespace where the closure was created, even though the enclosing function has returned and finished executing.

To define a closure, you need to take three steps:
1. Create an inner function.
2. Reference variables from the enclosing function.
3. Return the inner function.

Closures

```
def outer_func(x):
    y = 4
    def inner_func(z):
        print(f"x = {x}, y = {y}, z = {z}")
        return x + y + z
    return inner_func

for i in range(3):
    closure = outer_func(i)
    print(f"closure({i+5}) = {closure(i+5)}")
```

Output:

```
x = 0, y = 4, z = 5
closure(5) = 9
x = 1, y = 4, z = 6
closure(6) = 11
x = 2, y = 4, z = 7
closure(7) = 13
```

SQLabs
המכללה הגבוהה להייטק

decorators

Decorators in Python are a very powerful and useful tool that allows you to modify the behavior of a function or class. Think of decorators as wrappers that add some extra functionality to the function or method they are decorating, without permanently modifying the original function's behavior.

How Decorators Work

A decorator is essentially a function that takes another function as an argument, adds some kind of functionality, and returns another function without altering the source code of the original function being decorated.

decorators

An example of a decorator that logs messages before and after a function

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_hello():
    print("Hello!")

# Decorate the function
say_hello = my_decorator(say_hello)

say_hello()
```

Output:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

SQLabs
המכללה הגבוהה להייטק

decorators

The @ Syntactic Sugar

Python provides a syntactic sugar to use decorators in a simpler way using the @ symbol, which is placed above the definition of the function to be decorated.

The previous example can be rewritten as:

```
@my_decorator
def say_hello():
    print("Hello!")


say_hello()
```

This does exactly the same thing as manually decorating the function (**say_hello = my_decorator(say_hello)**), but it's cleaner and more readable.

decorators

Decorators with Arguments

Sometimes, you might want your decorator to accept arguments. This can be achieved by adding another level of function nesting:

```python
def repeat(num_times):
    def decorator_repeat(func):
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator_repeat

@repeat(num_times=3)
def greet(name):
    print(f"Hello {name}")

greet("World")
```

Output:

```
Hello World
Hello World
Hello World
```

decorators

Use Cases for Decorators

- Logging
- Access control and authentication
- Modifying or augmenting function behavior
- Registering functions (e.g., for web route handling)

Key Points to Remember

Decorators wrap a function, modifying its behavior.
The @ symbol provides a cleaner way to decorate functions.
Decorators can accept arguments by adding another level of function nesting.
Though powerful, decorators should be used judiciously to keep the code understandable.
Decorators leverage the concepts of first-class functions and closures in Python, providing a flexible way to add functionality to your code dynamically.

CLASS DECORATOR

A class decorator adds a class to a function, and it can be achieved without modifying the source code.

For example, a function can be decorated with a class that can accept arguments or with a class that can accept no arguments

The __call__ method enables Python programmers to write classes where the instances behave like functions and can be called like a function

The __call__ method

```
class Product:
        def __init__(self):
                print("Instance Created")

        def __call__(self, a, b):
                print(a * b)

ans = Product()
ans(10, 20)
```

CLASS DECORATOR

```
class Power(object):
    def __init__(self, arg):
        self._arg = arg

    def __call__(self, a, b):
        retval = self._arg(a, b)
        return retval ** 2
@Power
def multiply_together(a, b):
        return a * b
```

Opening files

The buit-in open() function returns a file object
and is most commonly used with two arguments:
open(**filename**, mode)

Filename path can be relative to the working directory:

```
f = open("test_folder/test.py")
```

Or absolute path:

```
f = open("C:/Users/melni/Desktop/pythonProject5/test.xlsx")
```

File access modes

Mode	Name	Handle position	If file does not exist	For existing file
"r"	Read Only	beginning	raises I/O error	Read only
"r+"	Read and Write	beginning	raises I/O error	overwrite
"w"	Write Only	beginning	Creates the file	truncate and overwrite
"w+"	Write and Read	beginning	Creates the file	truncate and overwrite
"a"	Append Only	end	Creates the file	insert data at the end
"a+"	Append and Read	end	Creates the file	insert data at the end

Writing to a file

There are two ways to write in a file:

write() : Inserts the text in a single line in the text file

```
f = open("david.txt","w+")
f.write("hello")
```

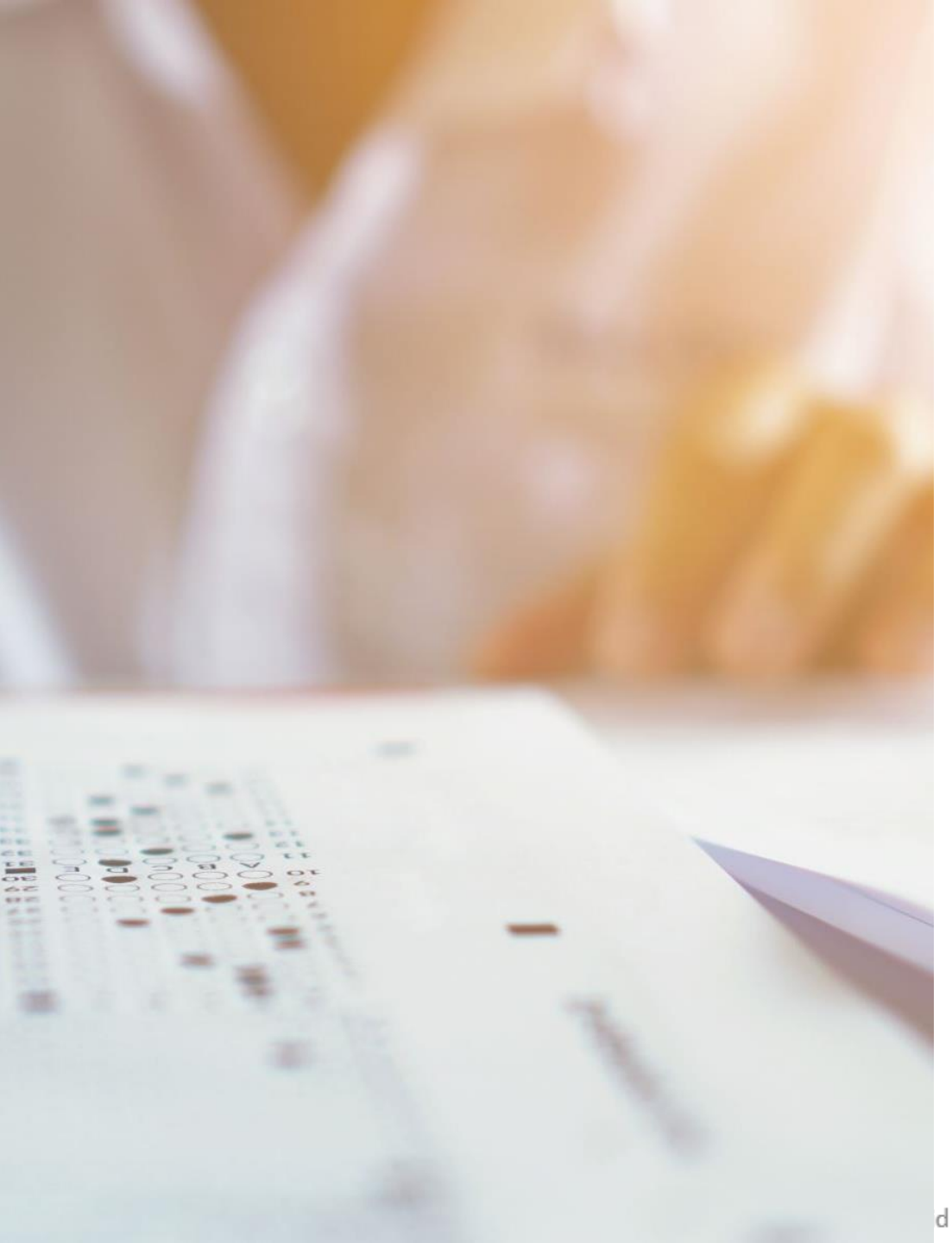
writelines() : For a list of string elements, each string is inserted in the text file. Used to insert multiple strings at a single time

```
f = open("david.txt","w+")
f.writelines(["hello\n","this is line two\n","more text"])
```

file close()

It is a good practice to always close the file when you are done with it

```
f = open("david.txt","r")
# do something
f.close()
```



with command

The **with** statement is used for resource management and exception handling. It is usually used when working with files.

```
with open('file-path', 'w') as file:
    file.write('Lorem ipsum')
```

Reading the file with **read**()

```
f = open("david.txt","r")
print(f.read())
```

Output:
hello
this is line two
more text

By default the **read**() method returns the whole text, but you can also specify how many characters you want to return

```
f = open("david.txt","r")
print(f.read(12))
```

Output:
hello
this i

Reading the file with **readline()**

You can return one line by using the **readline()** method

```
f = open("david.txt","r")
print(f.readline())
```

Output:
hello

By looping through the lines of the file, you can read the whole file, line by line:

```
f = open("david.txt","r")
for x in f:
    print(x, end="")
```

Output:
hello
this is line two
more text

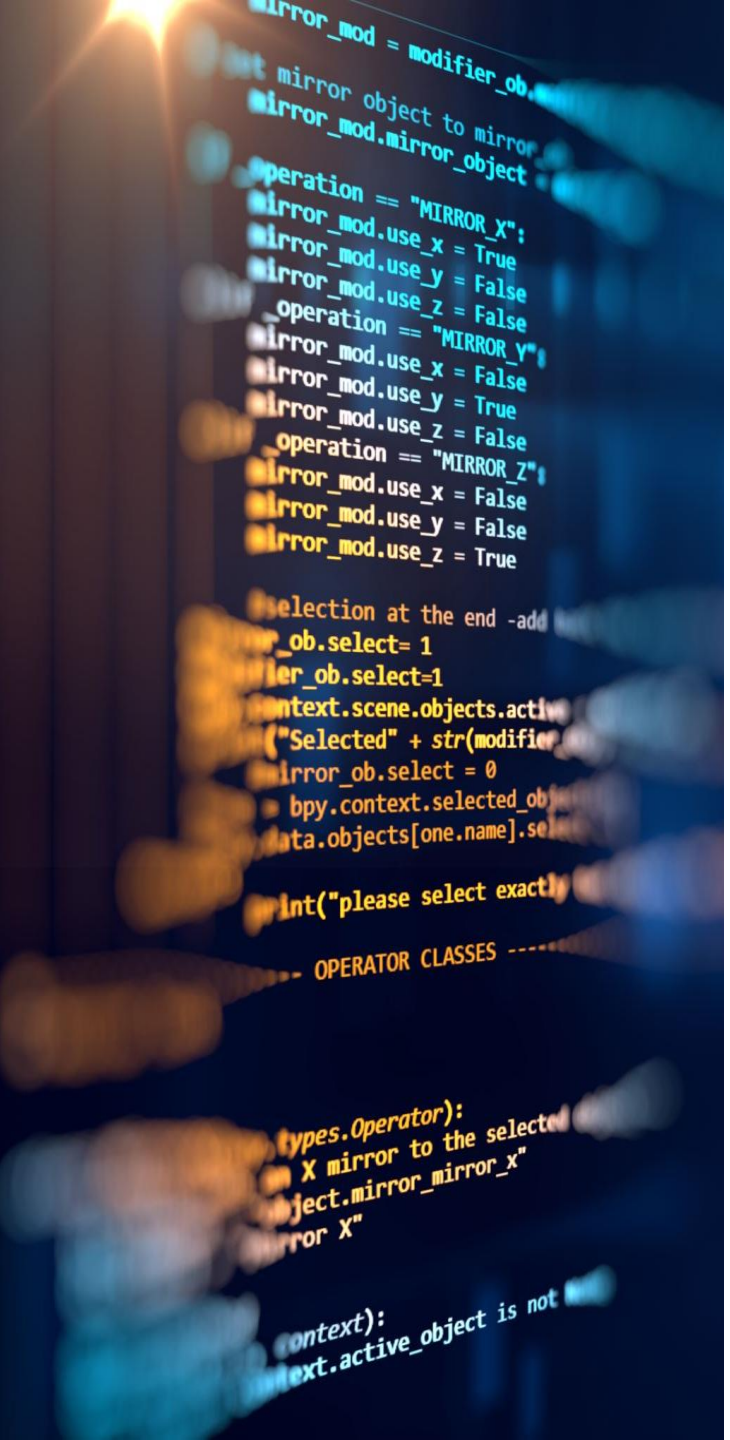
SQLabs
המכללה הגבוהה להייטק

The **Seek** method

seek(offset, whence)
Use it to change the **File Handle** position. **Offset** indicates the number of bytes to be moved. **whence** indicates from where the bytes are to be moved (default for the beginning).
For example, sample.txt has the following text:
Code is like humor. When you have to explain it, it's bad.

```
f = open("Practices/sample.txt", "r")
f.seek(20)
print(f.readline())
```

Output:
When you have to explain it, it's bad.



Practice - text files

1. Write a program to create a file on the desktop called myFile.txt and write on it the content 'Python is great'.
2. Write a program to write a list to a file, and then print the file content
3. Write a program to copy the contents of a file to another file
4. Write a program to count the number of lines in a text file

Practice - text files

1. Write a program to merge 2 text files into a new text file, and delete the original files
2. Write a program to merge all text files under the current directory into 1 file
3. Create my_folder (only if it does not exist), in my folder, create a text file: desktop.txt, in the desktop.txt create a list of all the files (only files, not folders) that exists in your desktop

CSV File Reading and Writing

A CSV file (Comma Separated Values file) is a type of plain text file that uses specific structuring to arrange tabular data

```
import csv
```

The **reader** object

Reading from a CSV file is done using the **reader** object.

```
import csv
f = open('example.csv')
reader = csv.reader(f)
```

The CSV file is opened as a text file with Python's built-in **open**() function, which returns a file object.

This is then passed to the reader object.

The **reader** object

```
import csv
f = open(sample.csv')
reader = csv.reader(f)
for row in reader:
    print(row[0], row[2])
```

Output:
First Name Location
David Israel
John UK
Jane Canada

The **DictReader** object

The **DictReader** can be used to read a CSV file as a dictionary

```
import csv
f = open('sample.csv')
reader = csv.DictReader(f)
print(f)
for row in reader:
    print(row['First Name'], row['Last Name'])
```

Output:

David Melnik

John Doe

Jane Da

The **writer** object

write to a CSV file using a **writer** object and the **write_row**() method

```
import csv
import random
csvfile = open('example.csv', 'w', newline='')
writer = csv.writer(csvfile)
writer.writerow(['Column A', 'Column B'])
for i in range(1,10):
    writer.writerow([random.randrange(1,100), random.randrange(1,100)])
```

SQLabs
המכללה הגבוהה להייטק

The DictWriter object

DictWriter maps dictionaries into CSV rows, using the **writerow**() command. The **fieldnames** parameter is a list of header names

```
import csv
f = open('names.csv', 'w')
fnames = ['first_name', 'last_name']
writer = csv.DictWriter(f, fieldnames=fnames)
writer.writeheader()
writer.writerow({'first_name': 'John', 'last_name': 'Smith'})
writer.writerow({'first_name': 'Robert', 'last_name': 'Brown'})
f.close()
```

The `writeheader()` method writes the headers to the CSV file

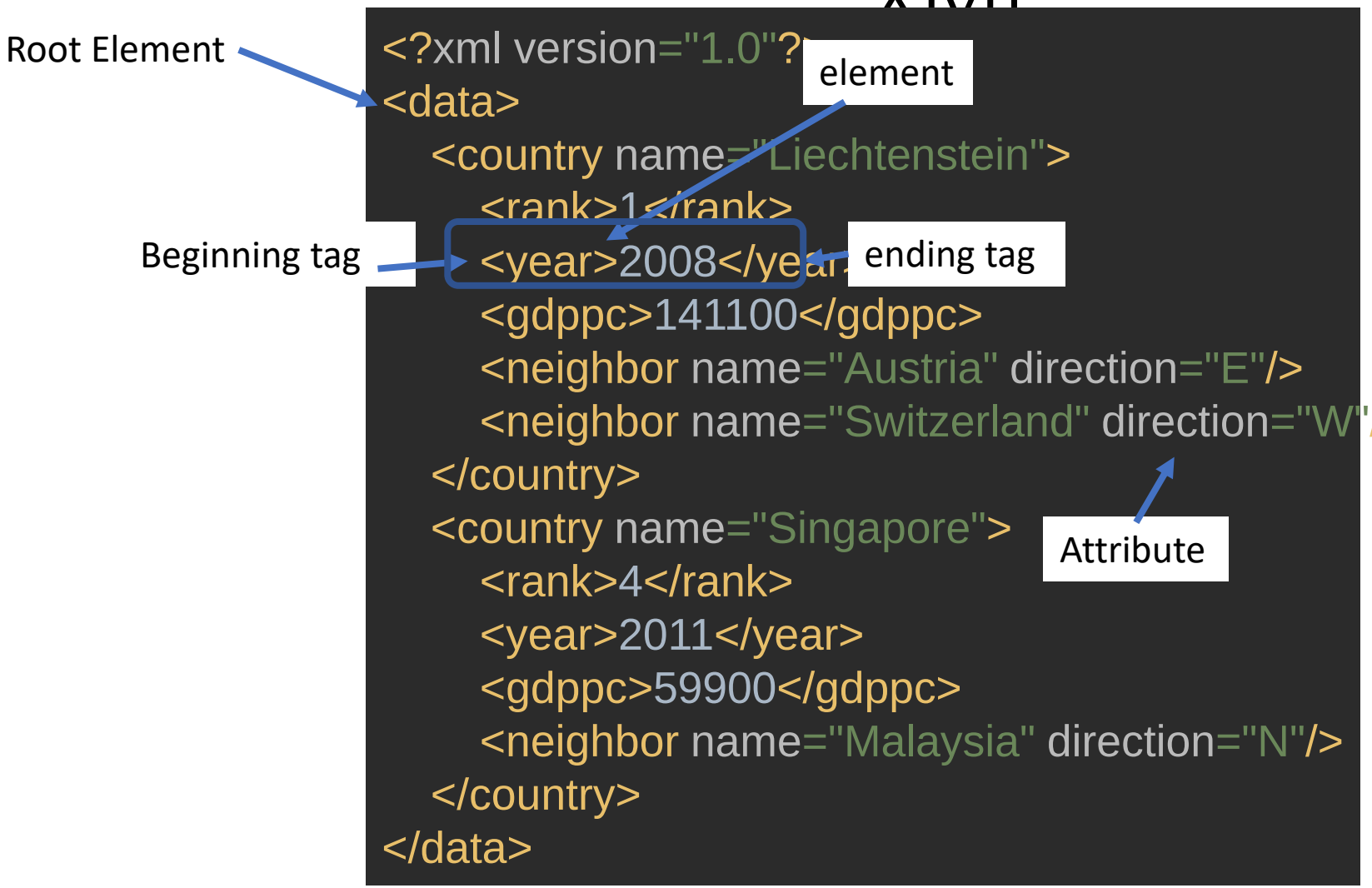
Practice - CSV

1. write a program to create dice.csv with columns: dice1, dice2 and 20 rows of random numbers between 1-6
2. Write a program to read from dice.csv and print the average of all the numbers in columns dice1 and dice2 (40 numbers)

XML

XML stands for **eXtensible Markup Language**. It is mainly used to store and transport data.
It was designed to be both human- and machine-readable.
The design goals of XML emphasize simplicity, generality, and usability across the Internet

XML



SQLabs
המכללה הגבוהה להייטק

The ElementTree

```
import xml.etree.ElementTree as ET
```

This module helps us format XML data in a tree structure which is the most natural representation of hierarchical data. Element tree has the following properties:

Property	Description
Tag	It is a string representing the type of data being stored
Attrib	Consists of a number of attributes stored as dictionaries
Text	A text string having information that needs to be displayed

The **xml.etree.ElementTree** module

The **parse** () function parses XML file

```
import xml.etree.ElementTree as ET
mytree = ET.parse("country_data.xml
myroot = mytree.getroot()
for element in myroot:
    print(element.attrib['name'])
```

```
<data>
  <country name="Liechtenstein">
    <rank>1</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
  <country name="Singapore">
    <rank>4</rank>
    <year>2011</year>
```

The **getroot**() function returns the root element

SQLabs
המכללה הגבוהה להייטק

The **fromstring()**

```
import xml.etree.ElementTree as
ET
myXML = """
<table>
    <row>
        <column1>a</column1>
        <column2>b</column2>
    </row>
    <row>
        <column1>c</column1>
        <column2>d</column2>
    </row>
</table>
"""
myRoot = ET.fromstring(myXML)
print(myRoot.tag)
```

fromstring returns the root element of the XML.

Output:
table

element.iter

```
import xml.etree.ElementTree as ET
mytree = ET.parse("country_data.xml")
myroot = mytree.getroot()
for rank in myroot.iter("rank"):
    print(rank.text)
```

```
<data>
  <country name="Liechtenstein">
    <rank>1</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
  <country name="Singapore">
    <rank>4</rank>
    <year>2011</year>
```

SQLabs
המכללה הגבוהה להייטק

xpath

- **XPath** stands for **XML Path Language**
- **XPath** uses "path like" syntax to identify and navigate nodes in an XML document
- **XPath** contains over 200 built-in functions

```
for rnk in myroot.findall(".//rank"):
    print(rnk.text)
```

xpath

findall returns a list of all element using a criteria
The findall("country") returns a list of country elements which exists directly below the root element

```
for country in myroot.findall("country"):
    print(country.attrib['name'])
```

Output:
Liechtenstein
Singapore
Panama

The findall(".//rank") returns a list of rank elements in any level below the root

```
for rnk in myroot.findall(".//rank"):
    print(rnk.text)
```

Output:
1
4
68

SQLabs
המכללה הגבוהה להייטק

xpath

Example:
In order to find which country is neighbor with 'Switzerland'
Select the parent of a **neighbor** element which as a value of "'Switzerland" in it's **name** attribute

```
for country in myroot.findall("country/neighbor[@name='Switzerland']/.."):
    print(country.attrib['name'])
```

Output:
Liechtenstein

JSON

JSON stands for **J**ava**S**cript **O**bject **N**otation
JSON is a lightweight format for storing and transporting data
JSON is often used when data is sent from a server to a web page
JSON is "self-describing" and easy to understand
Python has a built-in package called **json**, which can be used to work with JSON data

```
import json
```

JSON

example of a possible JSON representation describing a person

```
{
"firstName": "John", "lastName": "Smith",
"age": 27,
"address": { "streetAddress": "21 2nd Street", "city": "New
York", "state": "NY", "postalCode": "10021-3100" },
"phoneNumbers": [ { "type": "home", "number": "212 555-
1234" }, { "type": "office", "number": "646 555-4567" }]
}
```

SQLabs
המכללה הגבוהה להייטק

JSON

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

JSON

```
import json
print(json.dumps({"a":None,"b":True,"C":list(
)}))
```

Output:

{"a": null, "b": true, "C": []}

JSON

json.**loads** receive json string and convert it to dictionary

```
import json
person = '{ "name":"John", "age":30, "city":"New York"}'
json_person = json.loads(person)
print(json_person["age"])
```

requests

- The requests module in Python is a powerful HTTP library designed to send all kinds of HTTP requests easily. It
- handle many tasks involved in HTTP requests, like sending data, adding headers, and much more.

```
pip install requests
```

Requests – http get

A GET request is used to retrieve data from a specified resource.

```
import requests
response = requests.get('https://api.github.com')
```

Response Object

When you make a request to a web server, the server responds with a lot of information. The requests module encapsulates this information in a Response object.

- response.status_code: The status code indicates the status of the request (200 for success, 404 for not found, etc.).
- response.text: The content of the response, in unicode.
- response.json(): If the response data is in JSON format, this method parses the JSON and returns a dictionary

Practice - requests

1. Create a program that will get 10 random users from randomuser.me/api
2. The program will create a csv file with the columns: first name, last name and email of each random user

Regular Expressions

- A *regular expression* is a pattern - a template - to be matched against a string.
- Matching a regular expression against a string either succeeds or fails.
- Regular expressions are widely used by many programs and languages
- The module **re** provides full support for regular expressions in Python

```
[^ ]*?@[^ ]*?\.[^ ]*
```

regular expressions

re Module

- re.search: search for the first occurrence if a pattern in a string
- re.sub: replaces all occurrences of a pattern in string
- re.split: Split string by the occurrences of pattern
- re.findall: for all matched patterns in string
- re.compile().

[^]*?@[^]*?\.[^]*

SQLabs
המכללה הגבוהה להייטק

Regular- expression characters

Character	The character meaning	Example
^	Match the beginning of the line	^a
$	Match the end of the line (or before newline at the end)	a$
.	Match any character (except newline)	… ^…$
[ ]	Character class	[aeiouAEIOU] [a-zA-Z0-9_] [^0-9]
|	Alternation	abc|123
()	Grouping	(abc)+
\	Quote the next metacharacter	^\.

Regular- expression characters - Cont'd

- There is the basic set of quantifiers characters:

Character	The character meaning	Example
*	Match 0 or more times	^ab*c$
+	Match 1 or more times	^[A-Z]+
?	Match 1 or 0 times	[.?!]?$
{n}	Match exactly n times	.{20}
{n,}	Match at least n times	^A.{20,}
{n,m}	Match at least n but not more than m times	^[0-9]{4,9}$

Regular- expression characters - Cont'd

- There is the extended set of Python characters:

Character	The character meaning	Example
\w	Match a "word" character (alphanumeric plus "_")	^\w{5}$
\W	Match a non-"word" character	^\W.*\W$
\s	Match a whitespace character	\s
\S	Match a non-whitespace character	^\S+$
\d	Match a digit character	\d$
\D	Match a non-digit character	^\D

re search

re.search - Scan through string looking for the first location where the regular expression pattern produces a match, and return a corresponding MatchObject instance

matchObj = re.search(pattern, string, flags=0)

pattern – regular expression

string – string to look *pattern* into

flags – possible flags

matchObj will be None if pattern didn't match

re flags

re.I	Performs case-insensitive matching.
re.M	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
re.S	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
re.X	Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats un-escaped # as a comment marker.

re Matching Object

- matchObj has the following attributes:
  - matchObj.group(i) - returns matched group number *i* (one-based)
  - matchObj.start(i) - returns the offset in the string of the start of group number *i*
  - matchObj.end(i) returns the offset of the character beyond the match of group number *i*

You can use the m.start() and m.end() to slice the string:
string[m.start(i):m.end(i)]

re search

```
import re
line = "my age is 22"
m = re.search(r "\d+",line)
if m:
    print(f"matched string is {m.group()} in index ({m.start()},{m.end()})")
else:
    print("No match!!")
```

```
matched string is 22 in index (10,12)
```

re search – cont'd

```
import re
line = "27:11:2004"
m = re.search(r'(\d+):(\d+):(\d+)', line)
if m:
    print (f"matched day is {m.group(1)} in index({m.start(1)},{m.end(1)})")
    print (f"matched month is {m.group(2)} in index({m.start(2)},{m.end(2)})")
    print (f"matched year is {m.group(3)} in index({m.start(3)},{m.end(3)})")
else:
    print ("No match!!")
```

```
matched day is 27 in index(0,2)
matched month is 11 in index(3,5)
matched year is 2004 in index(6,10)
```

re sub

- **re.sub** – replaces all (or max) occurrences of the pattern in string. This method would return modified string

- re.sub(pattern, repl, string, max=0)

  pattern – regular expression

  repl – replacement string

  string – string to look *pattern* into

  max – maximum replacements

re sub – cont'd

```
import re
phone = ""2004-959-559
 # Remove anything other than digits
newPhone = re.sub(r'\D', "", phone)
print ("Phone num now is : ", newPhone )

 #Replace '-' with space
newPhone = re.sub(r'-', " ", phone)
print ("Phone num now is : ", newPhone)
```

Phone num now is : 2004959559
Phone num now is : 2004 959 559

re split

- **re.split** - Split string by the occurrences of pattern

re.split(pattern, string, maxsplit=0, flags=0)

pattern – regular expression

string – string to look *pattern* into

maxsplit – maximum splits

flags – possible flags

re split – cont'd

```
import re
value ="one is 1, two is 2"
result = re.split("[, ]+", value)

for element in result:
    print(element)
```

Output:

```
one
is
1
two
is
2
```

re split – cont'd

```
value = "one 1 two 22 three 3"
result = re.split(r"\D+", value)
for element in result:
 print(element)
```

Output:
1
22
3

re finditer

- **re.finditer** - Return an MatchObject iterator for all matched patterns in string

re.finditer(pattern, string, flags=0)

pattern – regular expression

string – string to look *pattern* into

flags – possible flags

re.finditer – cont'd

```
text = "this is a long sentence with a lot of words"
for m in re.finditer(r"(\w+)", text):
    print(f'{m.start(1)}-{m.end(1)}:{m.group(1)}')
```

Output:
this:0-4
is:5-7
a:8-9
long:10-14
sentence:15-23
with:24-28
a:29-30
lot:31-34
of:35-37
words:38-43

Greedy Quantifiers

* (asterisk) - Matches 0 or more occurrences of the preceding element. It tries to match as many occurrences as possible.

Example: The pattern a* will match 'aaaa' in 'aaaaa’.

+ (plus) - Matches 1 or more occurrences of the preceding element. Like the asterisk, it tries to match as many occurrences as possible.

Example: The pattern a+ will match 'aaaa' in 'aaaaa’.

Greedy Quantifiers

? (question mark) - Matches 0 or 1 occurrence of the preceding element, making it optional. Still greedy, it will match one occurrence if possible.

Example: The pattern a? will match 'a' in 'aabc’.

{m,n} - Matches from m to n occurrences of the preceding element. If n is omitted, it matches at least m occurrences with no upper limit.

Example: The pattern a{2,4} will match 'aaaa' in 'aaaaa'.

Non-greedy or Lazy Matching

To make these quantifiers non-greedy (or lazy), you append a ? right after them. This changes their behavior to match as little of the string as possible.

*? - Matches 0 or more occurrences in a non-greedy way.

+? - Matches 1 or more occurrences in a non-greedy way.

?? - Matches 0 or 1 occurrence in a non-greedy way.

{m,n}? - Matches from m to n occurrences in a non-greedy way.

Non-greedy or Lazy Matching

```
import re
text = "abbbbbbc"

match = re.search("ab+", text)
print(match.group())  # Outputs: 'abbbbbb'

match = re.search("ab+?", text)
print(match.group())  # Outputs: 'ab'
```

Non-greedy or Lazy Matching

Consider scraping HTML content:

```
import re
html = "<div>Hello <b>World</b> and <b>Universe</b></div>"
match = re.search("<b>.*</b>", html)
print(match.group())  # Greedy matching
#Output: <b>World</b> and <b>Universe</b>

match = re.search("<b>.*?</b>", html)
print(match.group())  # Non-Greedy matching
#Output: <b>World</b>
```

re.compile

The re.compile function compiles a regular expression pattern into a regular expression object.

This object can then be used to match against text using its methods, such as match(), search(), findall(), and finditer().

Using re.compile is beneficial when you need to use the same expression multiple times in your code, as it separates the compilation process from the matching process, improving efficiency and readability.

re.compile

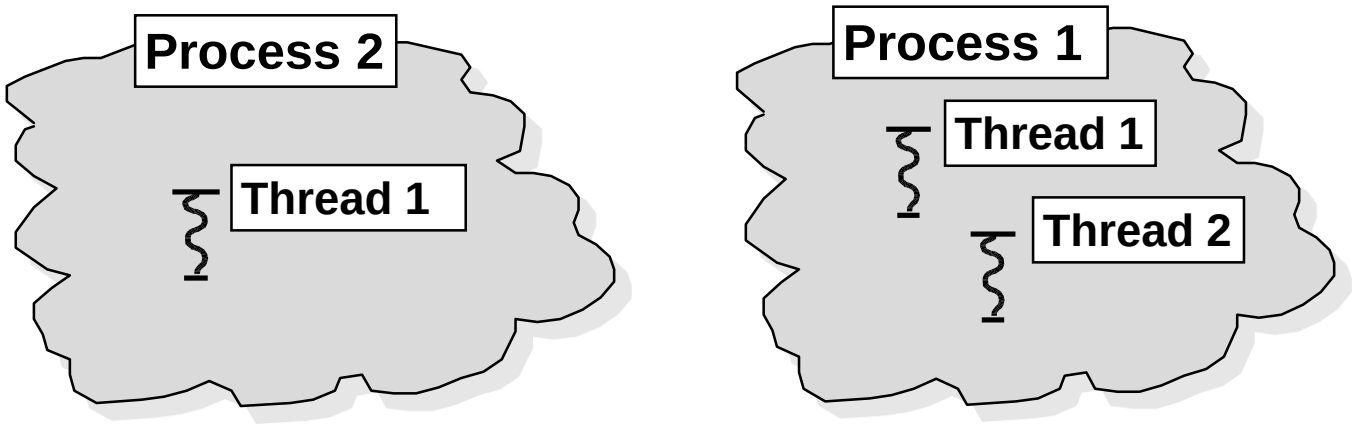
- Example of Using re.compile:
- import re
- email_pattern = re.compile(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,7}\b')
- def is_valid_email(email):
- if email_pattern.match(email):
- return True
- return False

- emails = ["user@example.com", "invalid-email", "another.user@domain.com"]
- for email in emails:
- print(f"{email}: {'valid' if is_valid_email(email) else 'invalid'}")

Parallelization

a **process** is an instance of a computer program that is being executed

A **thread** is an entity within a process that can be scheduled for execution
it is the smallest unit of processing that can be performed in an OS



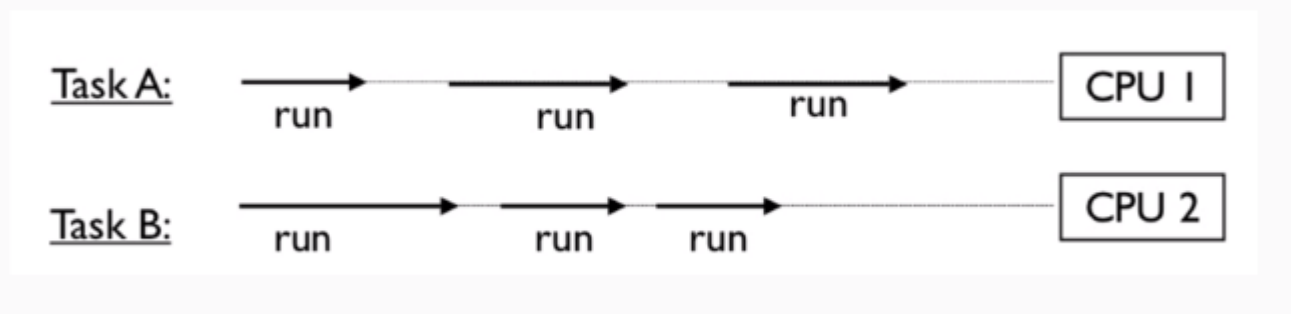
Parallelism Vs. Concurrency

Parallelism is performing multiple operations at the same time.
The **Multiprocessing** modules enables spreading tasks over a computer's central processing units (CPUs, or cores)

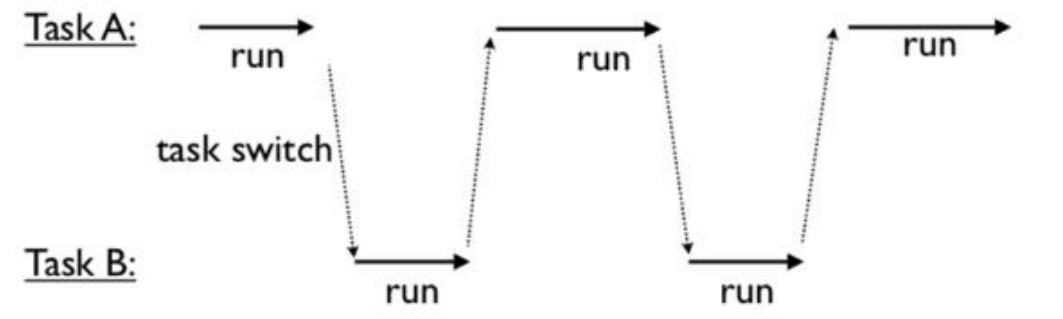
Concurrency suggests that multiple tasks have the ability to run in an overlapping manner.
The **Threading** module enables concurrency of multiple tasks.

Parallelism Vs. Concurrency

Parallelism



Concurrency



GLOBAL INTERPRETER LOCK

The Python **Global Interpreter Lock** or **GIL**, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter.

This means that only one thread can be in a state of execution at any point in time.

The impact of the GIL isn't visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code.

multiprocessing

multiprocessing module is responsible for processes creation and management

Python processes avoid the Global Interpreter Lock (GIL) and take full advantages of multiple processors on a machine

multiprocessing module includes a lot of useful classes for processes creation, synchronization and IPC

multiprocessing

We can use the **Process** class to create a process object

Process(**group**=None, **target**=None, **name**=None, **args**=())

- target is the callable object to be invoked by the Process
- name is the process name
- args is the argument tuple for the target invocation.
- group – should be always be None

multiprocessing

start() - Starts the process's activity.

join([timeout])

- Block the calling method until the process whose join() method is called terminates or until the optional timeout occurs.
- If timeout parameter specified the calling method will be blocked up to timeout seconds. The exitcode will stays None until the process is actually finishes

The Process example

```
from multiprocessing import *
def f(val):
    cp = current_process()
    print(f"in child process name={cp.name}, pid={cp.pid}, val = {val}")


if __name__ == '__main__':
    cp = current_process()
    print(f"in main process name={cp.name}, pid={cp.pid}")
    p = Process(target=f, args=('paramVal',))
    p.start()
```

Output:

```
in main process name=MainProcess, pid=11988
in child process name=Process-laVmarap = lav ,7512=dip ,1
```

multiprocessing

In some platform, like windows, the child process goes through the main space before executing the target function.

This is why the creation of child process must be under the:

if __name__ == '__main__':

Exchanging data between processes

When it comes to communicating between processes, the multiprocessing modules has two primary methods:

- Queues
- Pipes

multiprocessing– Queue

Multiprocessing Queue main functions:

- put
- get
- empty

multiprocessing– Queue

Multiprocessing Queue main functions:

- put
- get
- empty

multiprocessing– Queue

```
from multiprocessing import Process, Queue
def f(q):
    q.put(["one", "two",3])


if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints (["one", "two",3]
    p.join()
    q.close()
```

multiprocessing– Queue

- from multiprocessing import Process, Queue
- def f(q):
- q.put(["one", "two",3])

- if __name__ == '__main__':
- q = Queue()
- p = Process(target=f, args=(q,))
- p.start()
- print(q.get()) # prints (["one", "two",3]
- p.join()
- q.close()

multiprocessing– pipe

- Pipe() returns a pair of connection objects connected by a pipe which by default is duplex (two-way)
  - Each connection object has send() and recv() methods
- A Pipe can only have two endpoints
- A Pipe is much faster

multiprocessing– pipe

- from multiprocessing import Process, Pipe
- def f(conn):
- conn.send(['hello', 'world'])
- conn.close()

- if __name__ == '__main__':
- parent_conn, child_conn = Pipe()
- p = Process(target=f, args=(child_conn,))
- p.start()
- print(parent_conn.recv()) # ['hello', 'world']
- p.join()

multiprocessing– pool

- Multiprocessing module contains (among others) a Pool class that can be used for parallelize executing a function across multiple inputs.
- Using a Pool can be a convenient approach for simple parallel processing tasks. Some of Pool tasks are:
- pool.map
- pool.map_unordered
- pool.imap
- pool.map_async
- pool.apply

multiprocessing– pool

```
from multiprocessing import Pool
 def increment(number):
    return number + 1

if __name__ == '__main__':
    numbers = [1,2,3,4,5,6,7,8,9,10]
    pool = Pool(processes=3)
    incrementedList = pool.map(increment, numbers)
    print(incrementedList)          # [2,3,4,5,6,7,8,9,10,11]
```

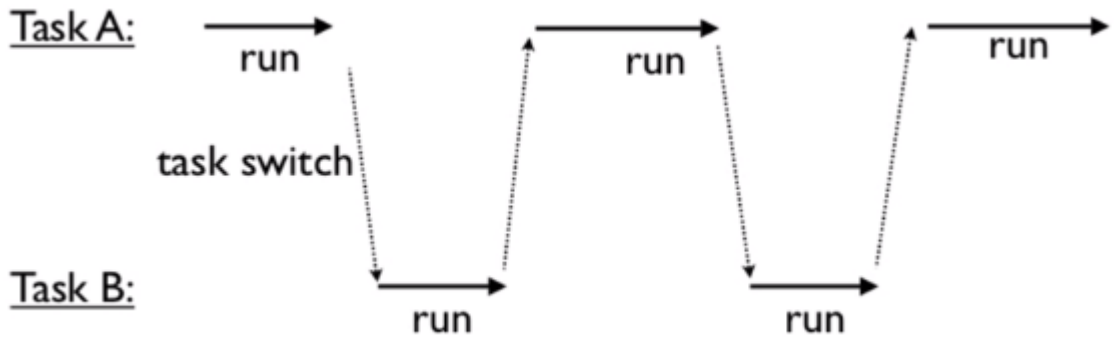
Multiprocessing Lock

- You can use a mutual exclusion (mutex) lock for processes via the multiprocessing.Lock class

- # create a lock
- lock = multiprocessing.Lock()
- # acquire the lock
- lock.acquire()
- # ...
- # release the lock
- lock.release()

Multithreading

One way to create concurrency is by using the **threading** module.
An example of concurrency:



Multithreading

To create a new thread, we create an object of **Thread** class.
It takes following arguments:

- **target**: the function to be executed by thread
- **args**: the arguments to be passed to the target function

To start a thread, we use **start** method of **Thread** class.

To ensure the finish of thread's job, we use **join** method of **Thread** class.

Multithreading

```
from time import sleep, time
def task():
    print('Starting a task...')
    sleep(1)
    print('done')

start_time = time()
task()
task()
end_time = time()
print(f'It took {end_time- start_time} second(s) to complete.')
```

Output:

```
Starting a task...
done
Starting a task...
done
It took 2.021 second(s) to complete.

Process finished with exit code 0
```

SQLabs
המכללה הגבוהה להייטק

Multithreading

```
from time import sleep, time
def task():
    print('Starting a task...')
    sleep(1)
    print('done')

start_time = time()
task()
task()
end_time = time()
print(f'It took {end_time- start_time} second(s) to complete.')
```

Output:
Starting a task...
done
Starting a task...
done
It took 2.021 second(s) to complete.

Process finished with exit code 0

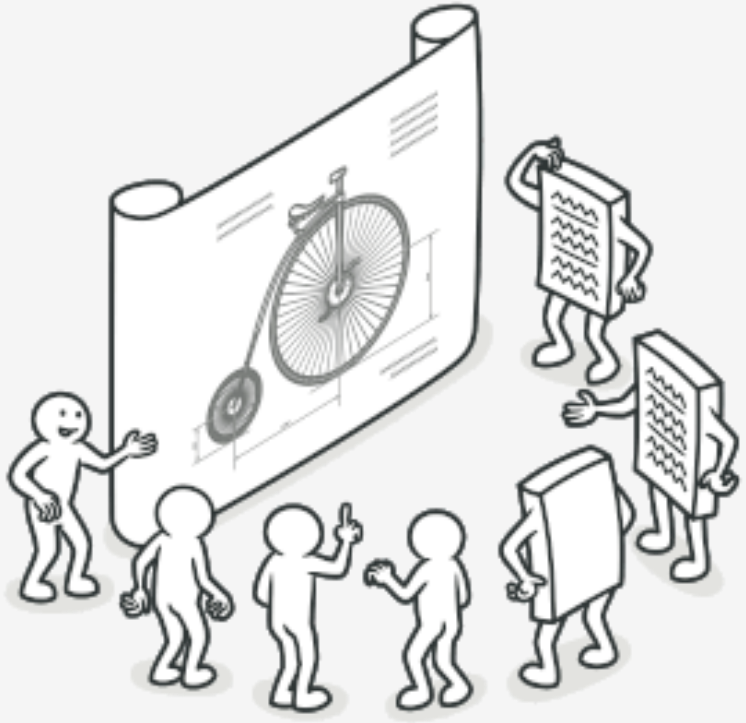
Multithreading

```
from time import sleep, time
from threading import Thread
def task(x):
    print('Starting a task...',x)
    sleep(1)
    print('done',x)

start_time = time()
t1 = Thread(target=task, args =
(1,))
t2 = Thread(target=task, args =
(2,))
t1.start()
t2.start()
print('something')
t1.join()
t2.join()
print('finish inner threads')
end_time = time()
print(f'It took {end_time- start_time:
0.2f} second(s) to complete.')
```

Design Patterns

Design patterns are typical solutions to common problems in software design.

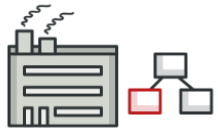


Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

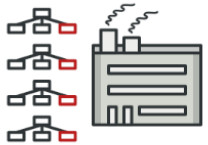
Types

Creational patterns

1

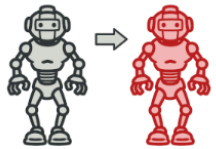


Factory Method

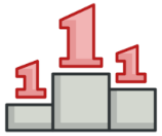


Abstract Factory

Builder



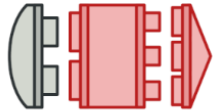
Prototype



Singleton

Structural patterns

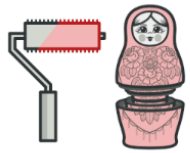
2



Adapter

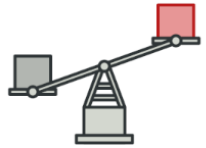
Bridge

Composite

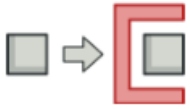


Decorator

Facade



Flyweight



Proxy

Behavioral patterns

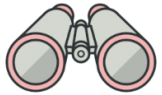
3

Chain of Responsibility

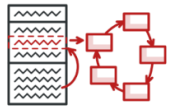
Command

Iterator

Memento



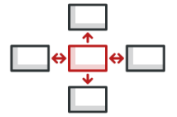
Observer



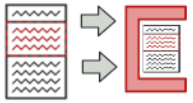
State

Template method

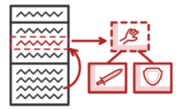
Visitor



Mediator

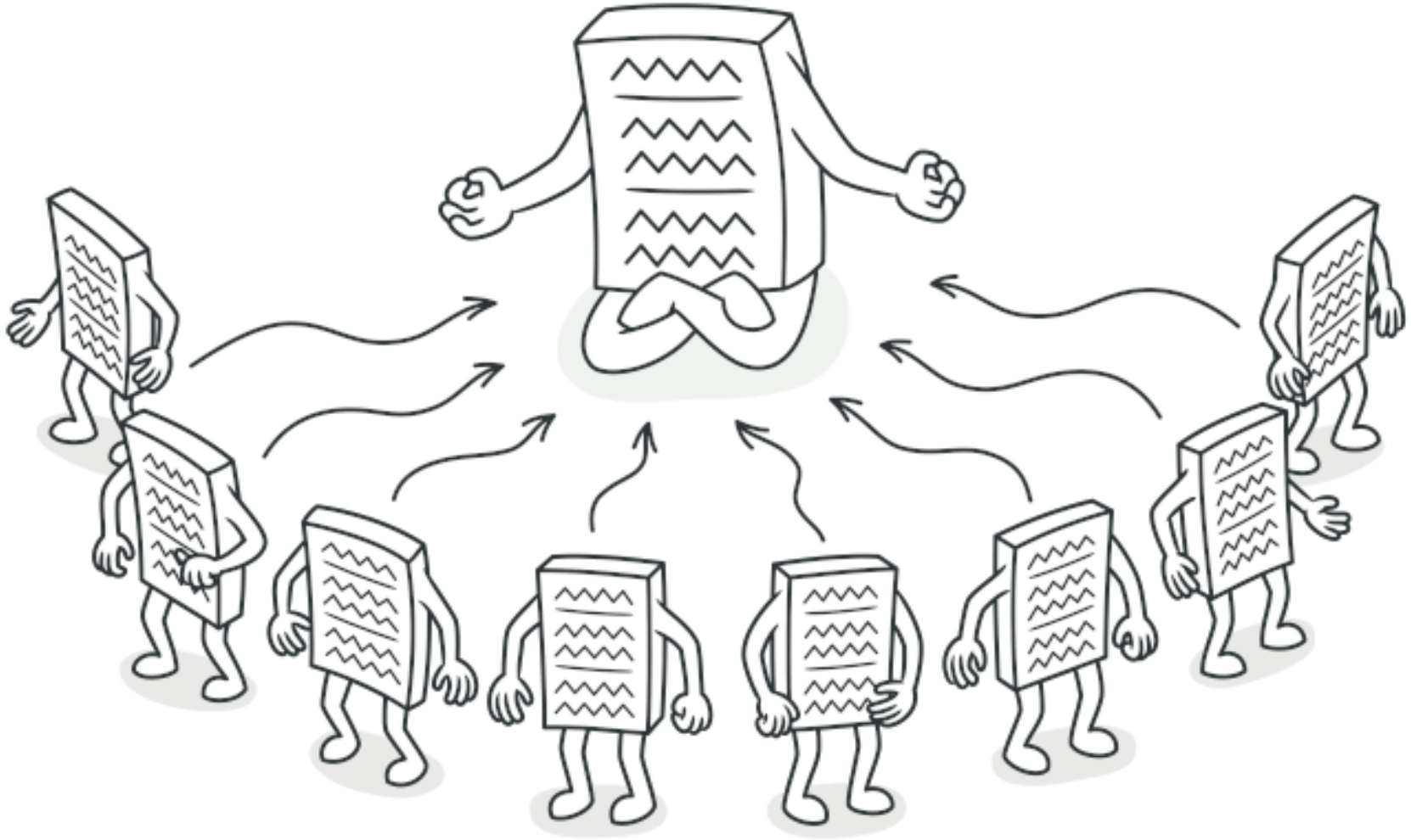


Interpreter



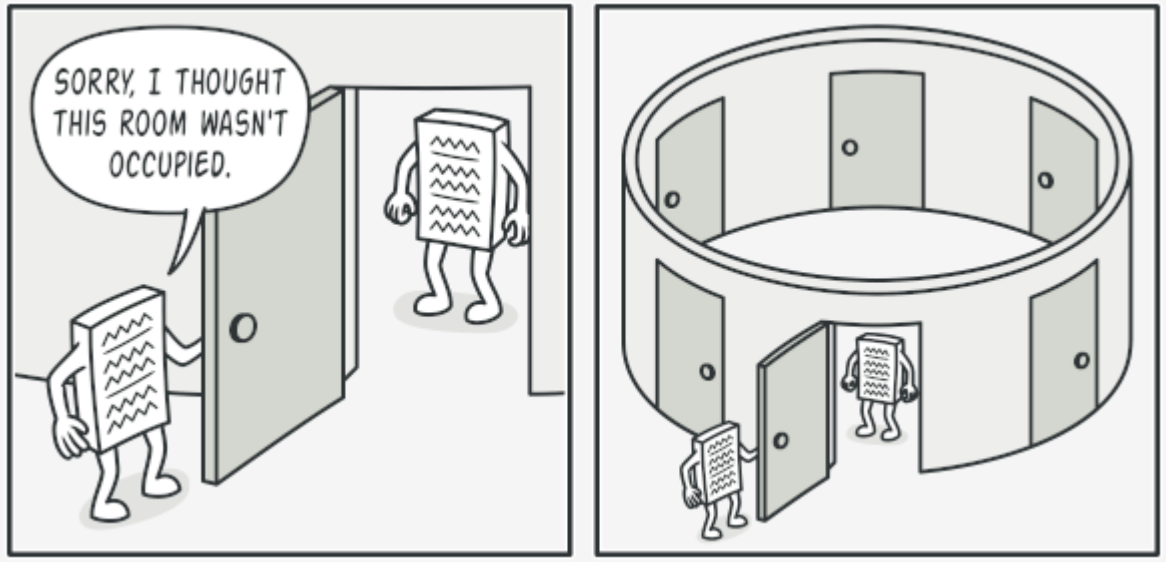
Strategy

singleton



Singleton

A singleton is a design pattern that restricts the instantiation of a class to one single instance.
This is useful when exactly one object is needed to coordinate actions across the system.
The Singleton pattern is often used in scenarios where system-wide actions need to be coordinated from a single point of operation, such as a database connection or a system configuration handler.



Singleton implementation 1

```
class Singleton:
    __instance = None

    def __new__(cls):
        if cls.__instance is None:
            cls.__instance = super().__new__(cls)
        return cls.__instance

singleton1 = Singleton()
singleton2 = Singleton()
print(singleton1 is singleton2)
```

Singleton implementation 2

```
class Singleton:
    __instance = None

    def __new__(cls):
        if cls.__instance is None:
            cls.__instance = super().__new__(cls)
            return cls.__instance
        else:
            raise Exception("An instance of the object already exist")

singleton1 = Singleton()
print("first")
singleton2 = Singleton()
print("second")
```

```
Exception: An instance of the object already exist
first
```

Singleton implementation 3

```
class Singleton:
    __instance = 0

    def __new__(cls):
        if cls.__instance < 3 :
            cls.__instance +=1
            return super().__new__(cls)
        else:
            raise Exception("the maximum number of objects (3)")
    def __init__(self):
        print("object created")

for i in range(4):
    Singleton()
```

```
Exception: the maximum number of objects (3)
object created
object created
object created
```

Singleton in the pythonic way

- A more Pythonic and reusable approach is to define a decorator that can turn any class into a singleton by wrapping its creation process.

Singleton implementation 4

```python
def singleton(cls):
    instances = {}
    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]
    return get_instance


@singleton
class Database:
    def __init__(self):
        self.connection = "Connection Established"
```

This decorator approach uses a dictionary to store instances and returns the same instance each time the class is instantiated.

Singleton implementation 5

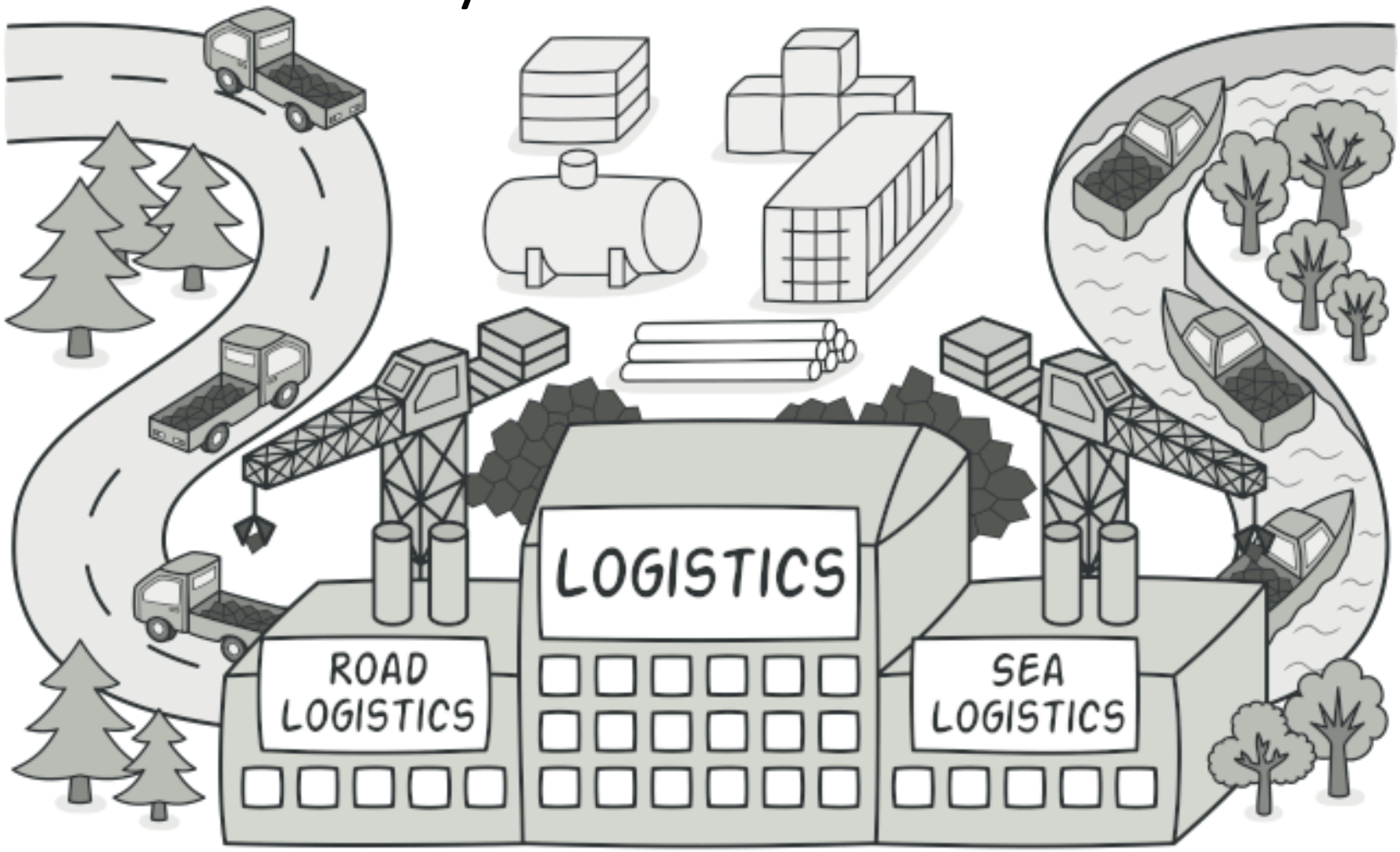
```python
class SingletonBase:
    _instances = {}

    def __new__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__new__(cls)
        return cls._instances[cls]

class Config(SingletonBase):
    def __init__(self):
        self.config = "Default Configuration"
```

Another approach is to use a base class that other classes can inherit to become singletons

Factory method



Factory method

The "Factory" design pattern is a popular and widely used design pattern in object-oriented programming.

It falls under the category of creational patterns, which are focused on the mechanism of creating objects in a way that increases the flexibility and reusability of code.

The Factory pattern is particularly useful when there is a need to create objects without specifying the exact class of object that will be created.

Factory method

Intent of the Factory Pattern: The primary intent of the Factory pattern is to define an interface for creating an object, but allow subclasses to alter the type of objects that will be created.

This pattern is used when a method returns one of several possible classes that share a common super class.

the Factory Design Pattern can be implemented using classes or functions.

Factory implemented using a class

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

    def speak(self):
        raise NotImplementedError()

# Concrete Products
class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

```python
class AnimalFactory:
    def create_animal(self, animal_type, name):
        if animal_type == "Dog":
            return Dog(name)
        elif animal_type == "Cat":
            return Cat(name)
```

```python
factory = AnimalFactory()
dog = factory.create_animal("Dog", "Buddy")
cat = factory.create_animal("Cat", "Whiskers")

print(dog.speak())  # Output: Woof!
print(cat.speak())  # Output: Meow!
```

Factory implemented using a function

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

    def speak(self):
        raise NotImplementedError()

# Concrete Products
class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

```python
def create_dog(name):
    return Dog(name)

def create_cat(name):
    return Cat(name)
```

```python
def create_animal(animal_type, name):
    if animal_type == "Dog":
        return create_dog(name)
    elif animal_type == "Cat":
        return create_cat(name)
```

```python
dog = create_animal("Dog", "Buddy")
cat = create_animal("Cat", "Whiskers")

print(dog.speak())  # Output: Woof!
print(cat.speak())  # Output: Meow!
```

Example of Factory Method Pattern

```python
from abc import ABC, abstractmethod

class User(ABC):
    @abstractmethod
    def get_role(self):
        pass

# Concrete Products
class Admin(User):
    def get_role(self):
        return "Admin role with all access"

class Guest(User):
    def get_role(self):
        return "Guest role with limited access"
```

Example of Factory Method Pattern

```python
class UserFactory(ABC):
    @abstractmethod
    def create_user(self, type):
        pass

# Concrete Creator
class SimpleUserFactory(UserFactory):
    def create_user(self, type):
        if type == "admin":
            return Admin()
        elif type == "guest":
            return Guest()
        else:
            raise ValueError("Unknown user type")
```

Example of Factory Method Pattern

```
if __name__ == "__main__":
    factory = SimpleUserFactory()
    user = factory.create_user("admin")
    print(user.get_role())  # Outputs: Admin role with all access

    user = factory.create_user("guest")
    print(user.get_role())  # Outputs: Guest role with limited access
```

Built-in Factory Method

```
f = open('a.txt', 'r')
print(type(f))

f2 = open('a.png', 'rb')
print(type(f2))
```

```
C:\Users\david\PycharmProjects\
<class '_io.TextIOWrapper'>
<class '_io.BufferedReader'>
```

multiprocessing

- Multiprocessing in Python refers to the ability to run multiple processes concurrently, taking advantage of multiple CPU cores or processors. This can significantly improve the performance of CPU-bound tasks by distributing the workload across multiple cores or processors. Python's built-in multiprocessing module provides a way to create and manage processes in a simple and efficient manner.

Multiprocessing - Process

```
import multiprocessing
import time

def sleepy_man():
    print('Starting to sleep')
    time.sleep(1)
    print('Done sleeping')

if __name__ == "__main__":
    tic = time.time()
    p1 = multiprocessing.Process(target=sleepy_man)
```

ividual process.

the multiprocessing

What is Thread

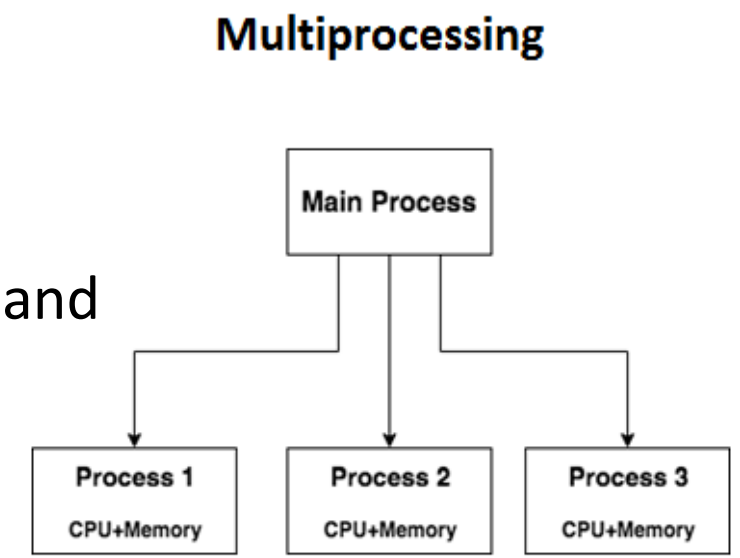
- A Thread or a Thread of Execution is defined in computer science as the smallest unit that can be scheduled in an operating system

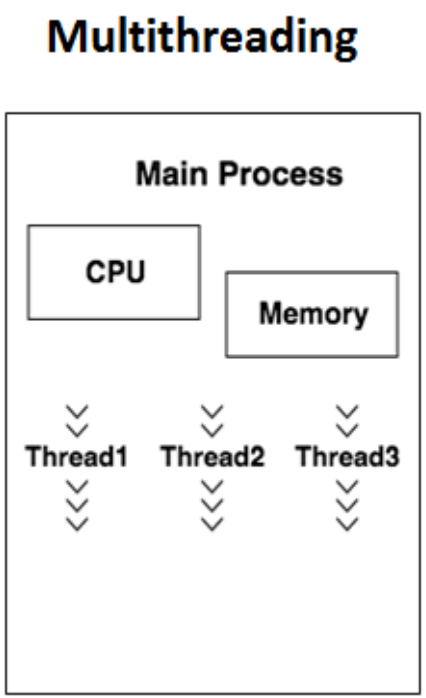
- Threads are contained in processes.

More than one thread can

exist within the same process.

- These threads share the memory and
- the state of the process.





SQLabs
המכללה הגבוהה להייטק

threading Module

- Threading module is a simple way to create threads. Threading APIs very similar to multiprocessing API

- Using threads allows a program to run multiple operations concurrently in the same process space.

- To create a new thread in out program we should use the Tread. class of Threading module

Thread class

- Thread(group=None, target=None, name=None, args=())
  - — target is the callable object to be invoked by the Process
  - — name is the process name
  - — **args** is the argument tuple for the target invocation.
  - — group — should be always be None

- Thread has start and join functions, exactly like Process does

Threading Module example

```
import time
import threading

global_num = 10
def func():
   global global_num
   Global_num = 11

thread1 = threading.Thread(target=func)
thread1.start()
thread1.join()
print(global_num)
```

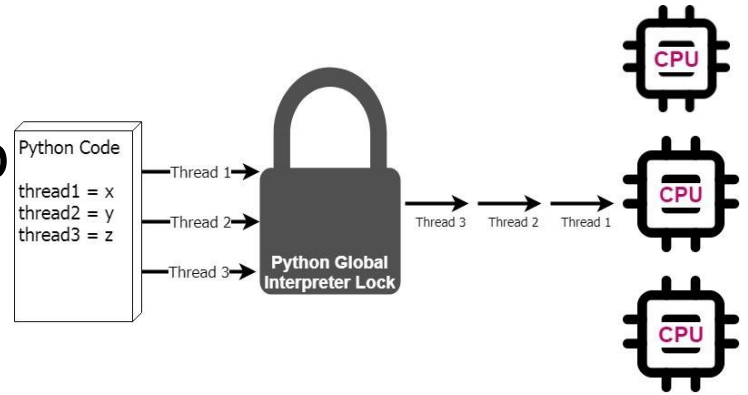
Threading module Synchronization

- threading module has 3 classes for threads synchronization, like multiprocessing module
  - Lock - non-recursive lock object
  - Rlock - recursive lock object
  - Semaphore — created with internal counter and can be acquired counter times before released

- lock = threading.Lock()
- with lock:
- # critical section code

The Global Interpreter Lock

- In CPython, the Global Interpreter Lock (GIL), is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once.
- The GIL is controversial because it prevents multithreaded CPython programs from taking full advantage of multiprocessor systems
- There are some GIL free operations,
- such as I/O and image processing. They happen o



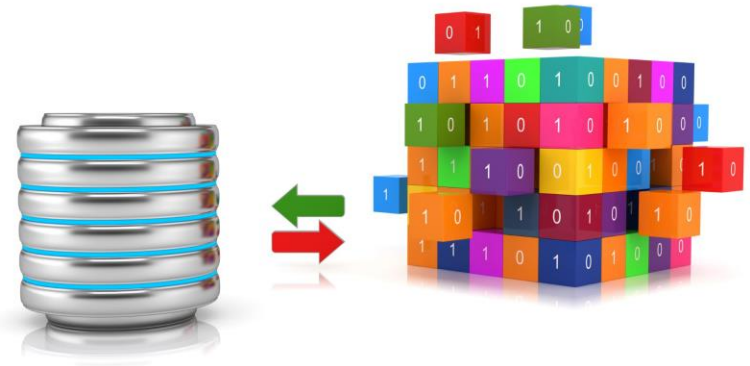
Introduction to SQLite3

Key Features of SQLite3

SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine.

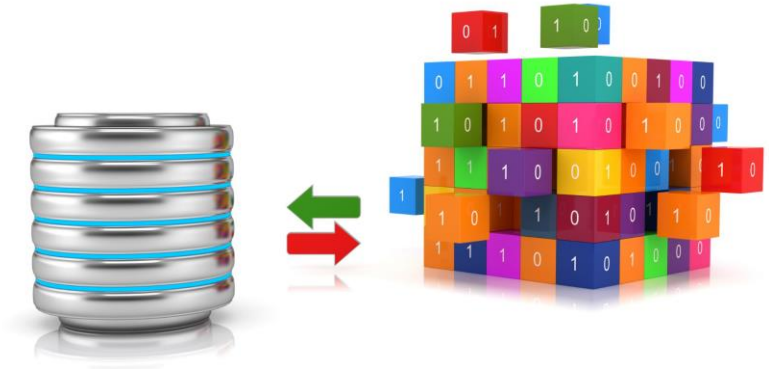
SQLite3 enables operations like connecting to a database, creating tables, inserting data, querying data, and handling transactions.

SQLite3 comes bundled with Python for versions 2.5 and above, so there's no need to install it separately.



Key Features of SQLite3

- Serverless architecture
- Zero configuration
- Stores the entire database as a single disk file
- Cross-platform support
- High performance for most read/write operations



SQLabs
המכללה הגבוהה להייטק

Connecting to a Database:

First, import the sqlite3 module and then establish a connection to the database file. If the file does not exist, SQLite will create it.

```
import sqlite3
conn = sqlite3.connect('mydatabase.db')
cursor = conn.cursor()
```

vith Python

Using SQLite3 with Python

Creating a table:

To create a table, you use the **CREATE TABLE** SQL statement and execute it using the execute() method of the cursor object.

```
cursor.execute('''
        CREATE TABLE users
        ([username] TEXT PRIMARY KEY, [password] TEXT)
        ''')
```

SQLabs
המכללה הגבוהה להייטק

Using SQLite3 with Python

Inserting Data:

Insert data into the table using the **INSERT INTO** SQL statement

```
cursor.execute("""INSERT INTO usersa
                  VALUES ('david','1234')""")
```

Using SQLite3 with Python

Committing Transactions:

Data changes are not actually saved into the database until you commit the transaction. You can commit the transaction by calling the commit() method on the connection object.

```
conn.commit()
```

Using SQLite3 with Python

Committing Transactions:

Data changes are not actually saved into the database until you commit the transaction. You can commit the transaction by calling the commit() method on the connection object.

```
conn.commit()
```

Using SQLite3 with Python

Querying Data:

Retrieve data by executing a SELECT statement. Then, use the cursor to fetch the results.

```
cursor.execute("""SELECT * FROM users
                  WHERE password='1234'""")
for row in cursor.fetchall():
    print(row)
```

SQLabs
המכללה הגבוהה להייטק

Using SQLite3 with Python

```
def valid_user(username, password):
    conn = sqlite3.connect('users.db')
    cur = conn.cursor()
    cur.execute(f"""select * from users
                where username = '{username}' and
                      password = '{password}'""")
    if cur.fetchone() == None:
        return False
    else:
        return True
```

Using SQLite3 with Python

Updating Data:

To change data already in the table, you use the UPDATE statement.

```
cursor.execute("""update users set password='admin'
                  WHERE username = 'admin' """)
```

Using SQLite3 with Python

Deleting Data:

To remove data from the table, use the DELETE statement.

```
cursor.execute("""delete from users
                  WHERE username = 'George' """)
```

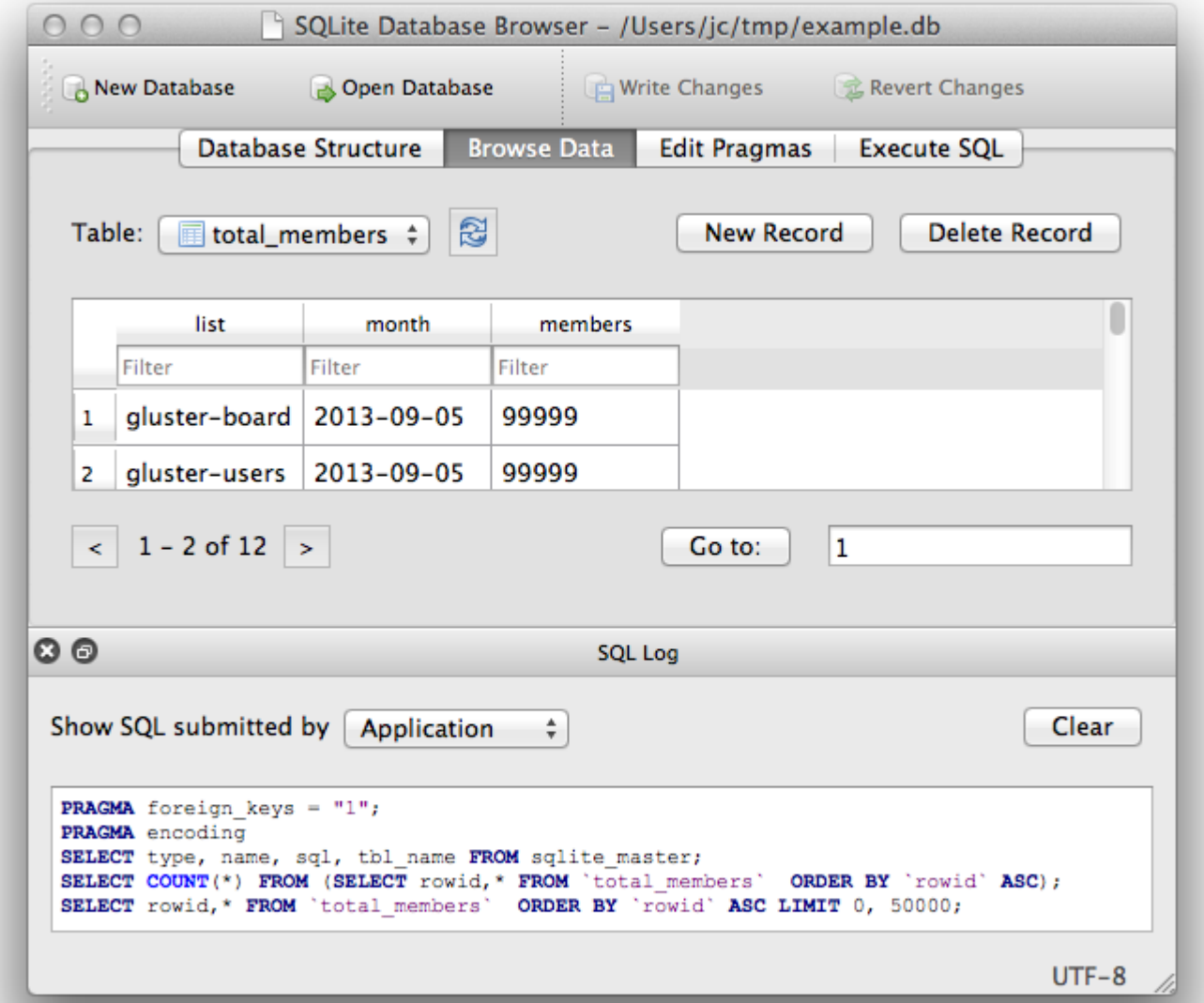
Using SQLite3 with Python

DB Browser for SQLite

DB Browser for SQLite is a high quality, visual, open source tool to create, design, and edit database files compatible with SQLite.

Downlod link:

sqlitebrowser.org



SQLabs
המכללה הגבוהה להייטק

Module: GUI frameworks

- Comparison of Popular Python GUI Frameworks
  - Tkinter, PyQt, etc.
- Basic Concepts: Windows, Widgets, and Event Handling
  - Creating a simple window and understanding the main event loop
- Introduction to Widget Types
  - Buttons, labels, text boxes, etc.
- Understanding Different Layout Managers
  - Grid, pack, and place in Tkinter
- Designing a User Interface
- Responsive Design
- Understanding Event-Driven Programming
- Implementing Real-Time Updates

POPULAR PYTHON GUI FRAMEWORKS

- **Tkinter**: The standard Python GUI library, included with most Python distributions. It can be used for developing windows, dialogs, buttons, menus, text fields, and many other GUI components

Qt is a framework for developing cross-platform applications

- **PyQt**: A set of Python bindings for the Qt framework (GPL license)
- **PySide**: A set of Python bindings for the Qt framework (LGPL license)

- **kivy**: an open-source framework for developing multi-touch applications. It can be used to create applications that run on Windows, Mac, Linux, Android, and iOS with the same codebase

BASIC CONCEPTS OF GUI FRAMEWORKS

Windows: The main container for the user interface, where all other elements are placed

Widgets: The basic building blocks of the user interface, such as buttons, text fields, and labels

Event Handling: The process of responding to user actions, such as button clicks and key presses

SQLabs
המכללה הגבוהה להייטק

תודה רבה!