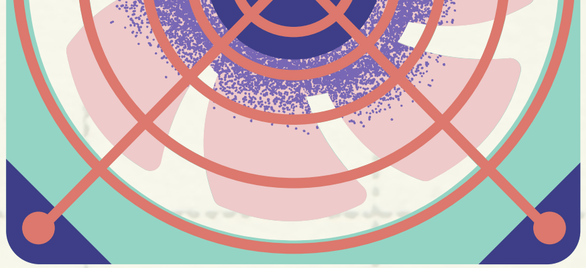
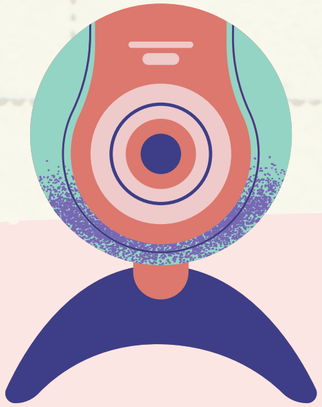
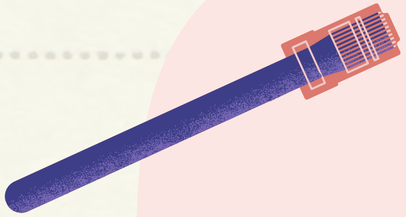
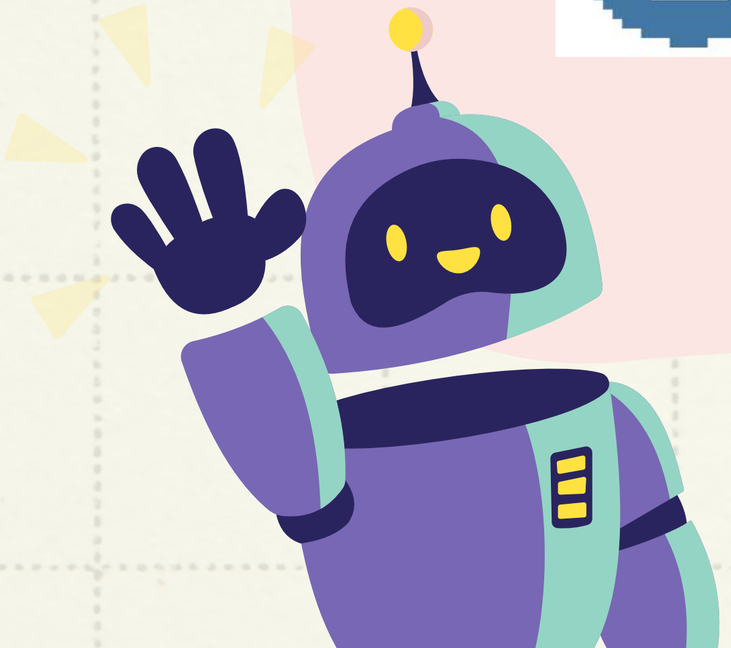


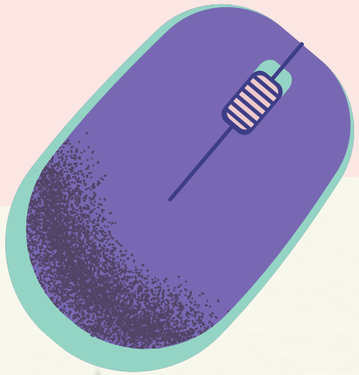


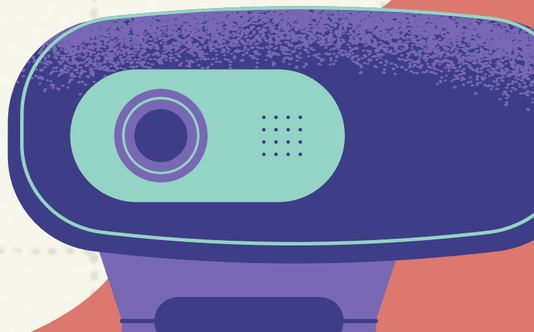


PYTHON

A BEGINNER'S GUIDE







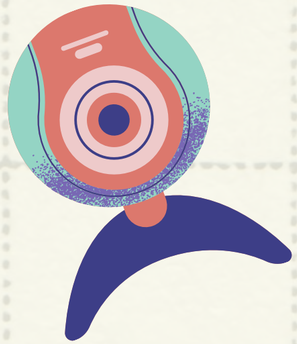
AGENDA

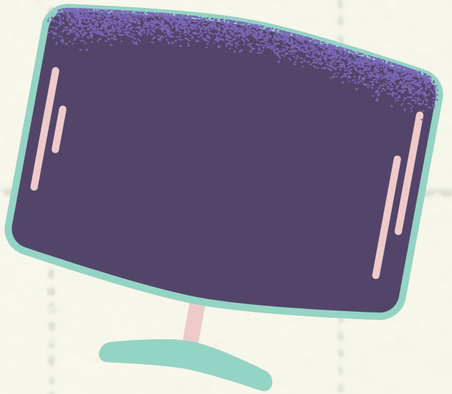
Data Types

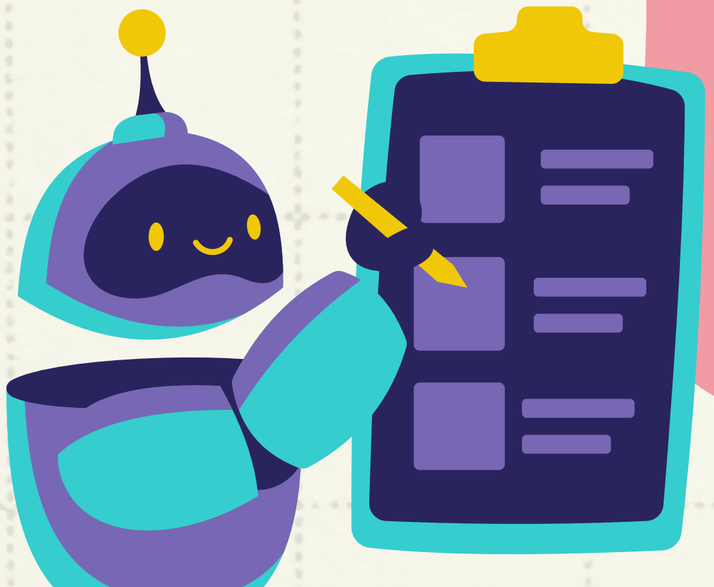
Data Structures

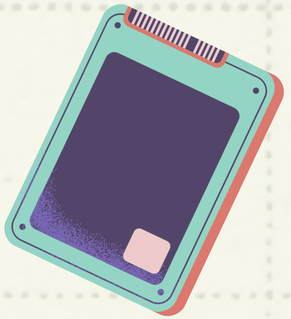
Functions

Classes

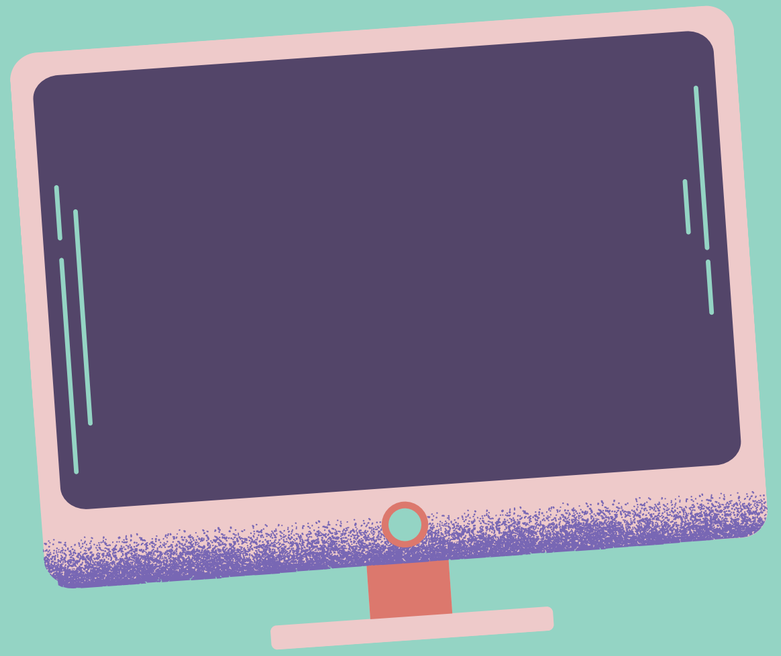


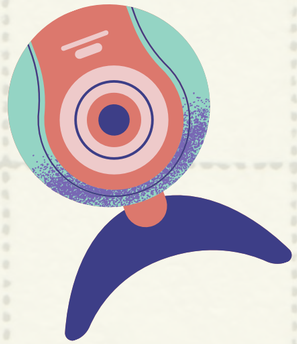


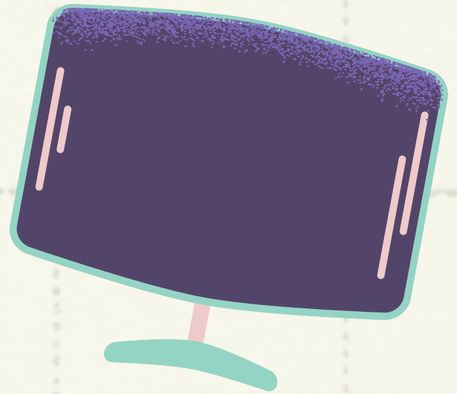




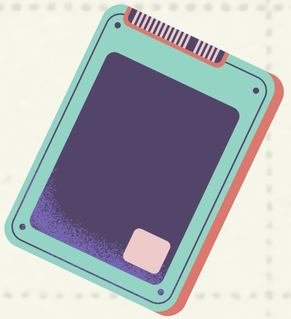
DATA TYPES



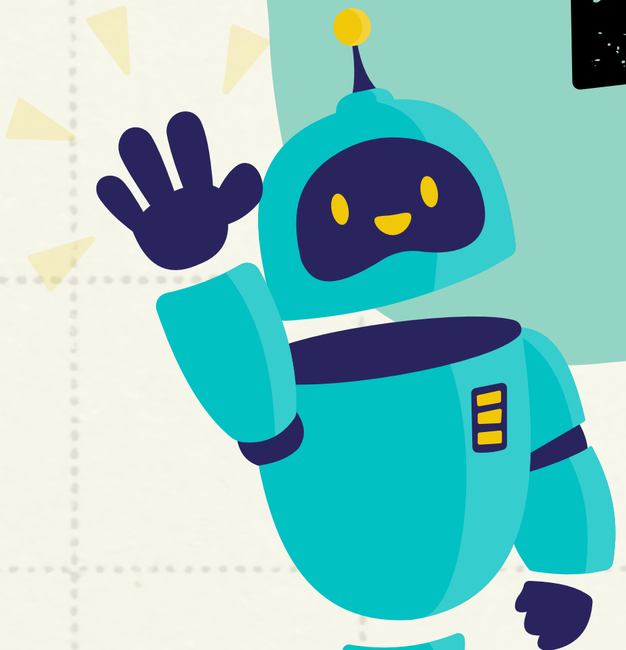


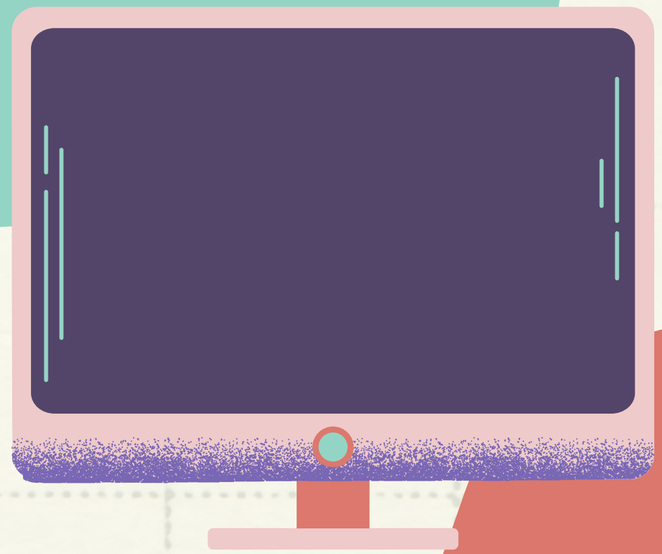


- String
- Int
- Float
- Complex
- Boolean
- None
- List



LISTS

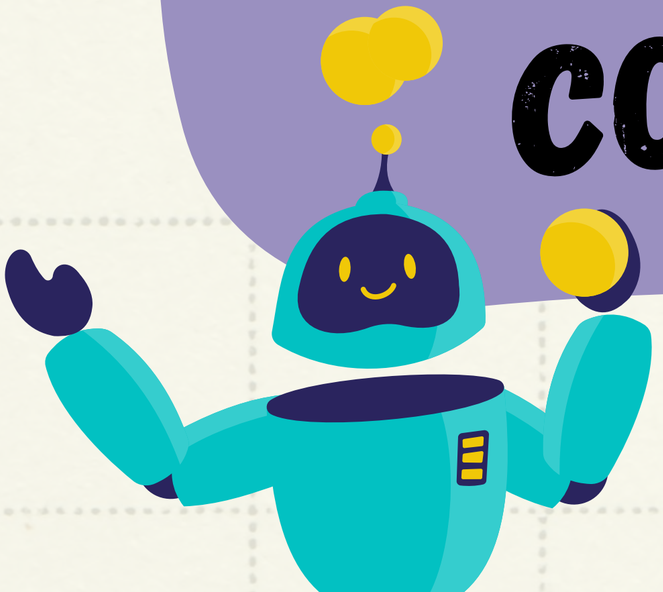




- LISTS ARE ORDERED COLLECTIONS OF ITEMS.
- CREATED USING SQUARE BRACKETS [] AND CAN HOLD ANY DATA TYPE.
- MY_LIST = [1, 2, 3, 'HELLO',TRUE]
- ACCESS ELEMENTS USING INDEX: MY_LIST[0] RETURNS ‘1’.

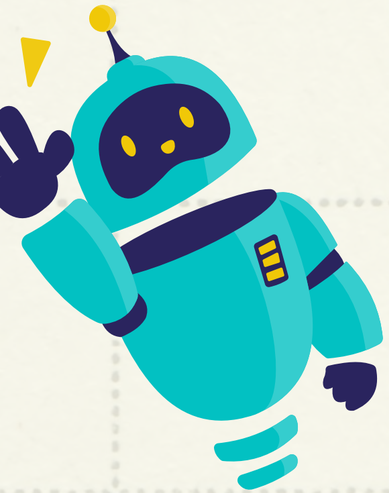
- APPEND: ADD AN ELEMENT TO THE END OF THE LIST
- MY_LIST.APPEND(4)
- REMOVE: REMOVE AN ELEMENT BY VALUE.
- MY_LIST.REMOVE('HELLO')
- LENGTH: GET THE NUMBER OF ELEMENTS.
- LENGTH = LEN(MY_LIST)

LIST SLICING AND COMPREHENSION

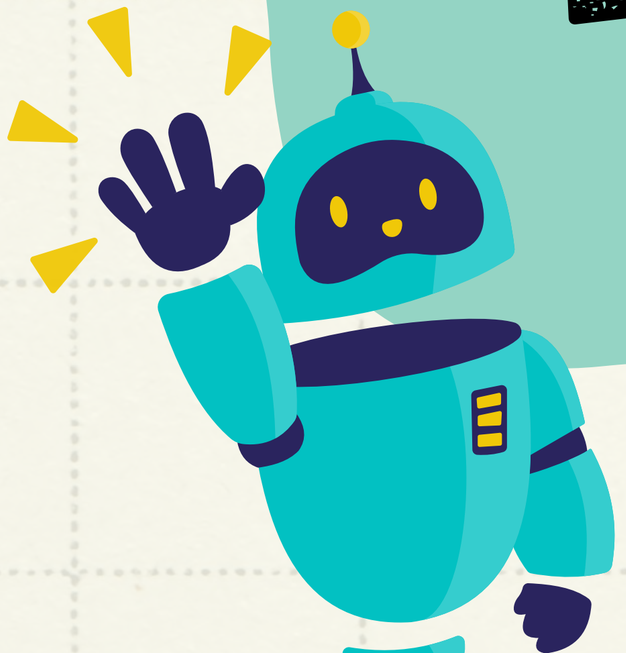


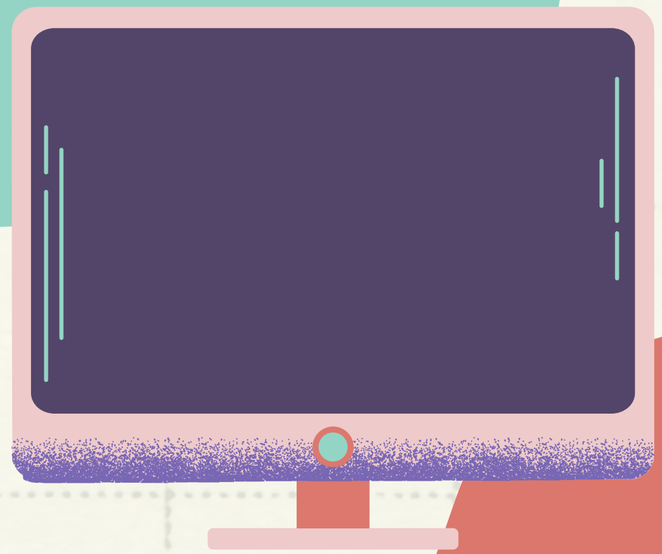
1. Access portions of a list using slicing..

- sliced_list = my_list[1:3]
- This creates a new list [2, 3] from **my_list**
-
- **Concise way to create lists from existing lists.**
- **Syntax: [expression for item in list if condition]**
- squares = [x**2 for x in range(1, 6)]
- his creates [1, 4, 9, 16, 25] by squaring
- each number from 1 to 5.



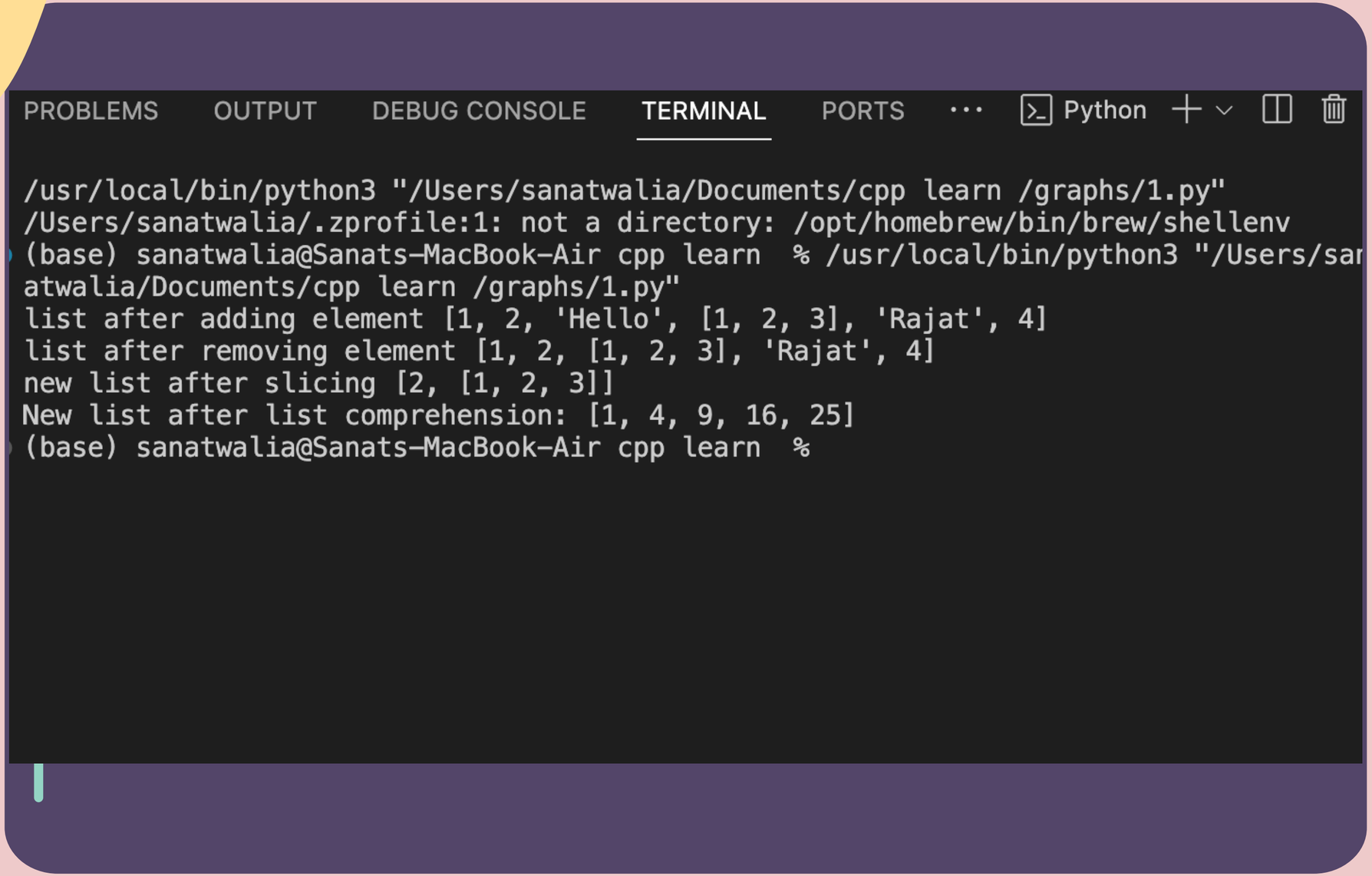
EXAMPLES



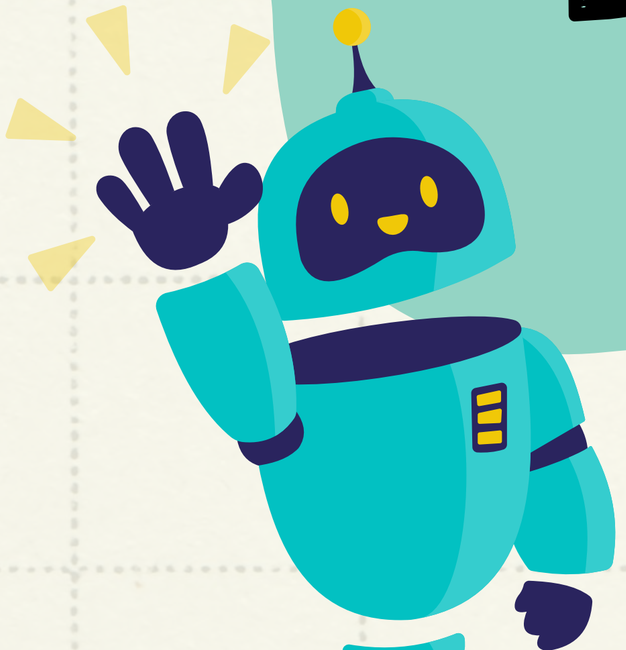


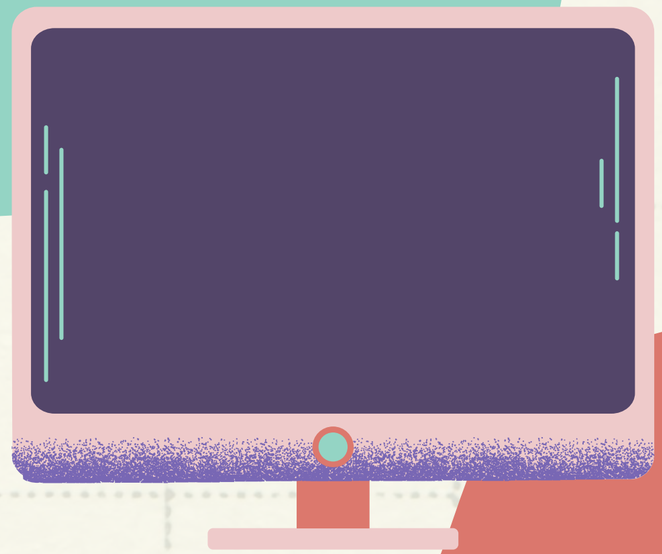
```python
my_list=[1,2,'Hello',[1,2,3],"Rajat"]
my_list.append(4)
print("list after adding element",my_list)
my_list.remove("Hello")
print("list after removing element",my_list)
sliced_list = my_list[1:3]
print("new list after slicing",sliced_list)

# list comprehension
squares = [x**2 for x in range(1, 6)]
print("New list after list comprehension:",squares)
```

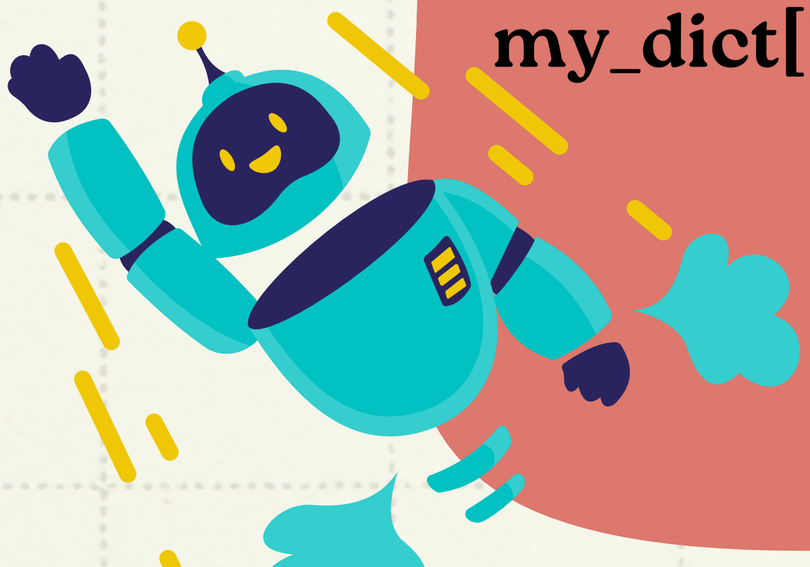

PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Python
/usr/local/bin/python3 "/Users/sanatwalia/Documents/cpp learn /graphs/1.py"
/Users/sanatwalia/.zprofile:1: not a directory: /opt/homebrew/bin/brew/shellenv
(base) sanatwalia@Sanats-MacBook-Air cpp learn % /usr/local/bin/python3 "/Users/sanatwalia/Documents/cpp learn /graphs/1.py"
list after adding element [1, 2, 'Hello', [1, 2, 3], 'Rajat', 4]
list after removing element [1, 2, [1, 2, 3], 'Rajat', 4]
new list after slicing [2, [1, 2, 3]]
New list after list comprehension: [1, 4, 9, 16, 25]
(base) sanatwalia@Sanats-MacBook-Air cpp learn %

DICTIONRY

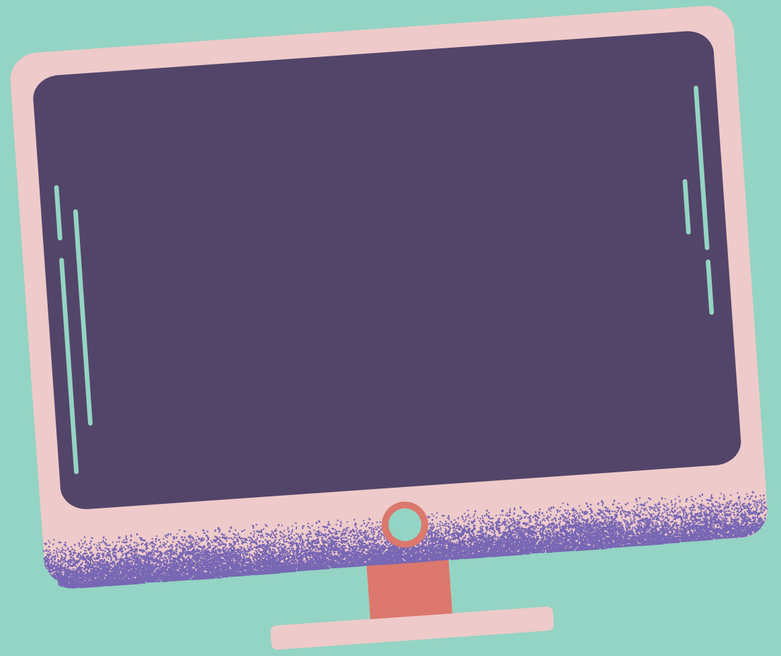




Dictionaries are unordered collections of key-value pairs.
Created using curly braces {} with keys and values separated by colons.
Access values by their keys:
my_dict['key'] returns 'value'



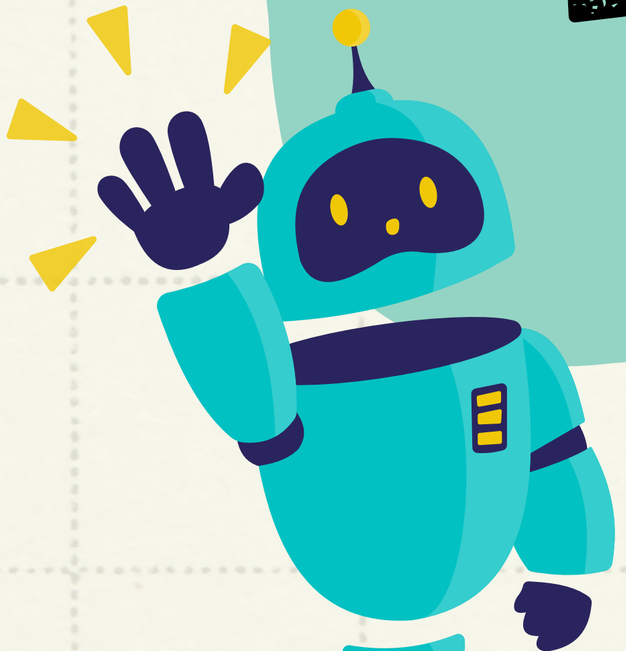
OPERATIONS

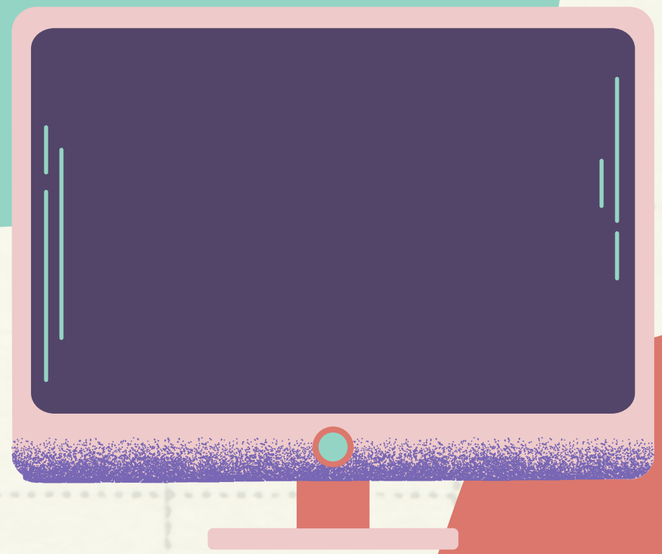


- Adding a Key-Value Pair:
- my_dict['occupation'] = 'Engineer'
- Removing a Key-Value Pair:
- del my_dict['age']
- Accessing Keys and Values:
- keys = my_dict.keys()
- values = my_dict.values()

- Adding a Key-Value Pair:
- my_dict['occupation'] = 'Engineer'
- Removing a Key-Value Pair:
- del my_dict['age']
- Accessing Keys and Values:
- keys = my_dict.keys()
- values = my_dict.values()

EXAMPLES

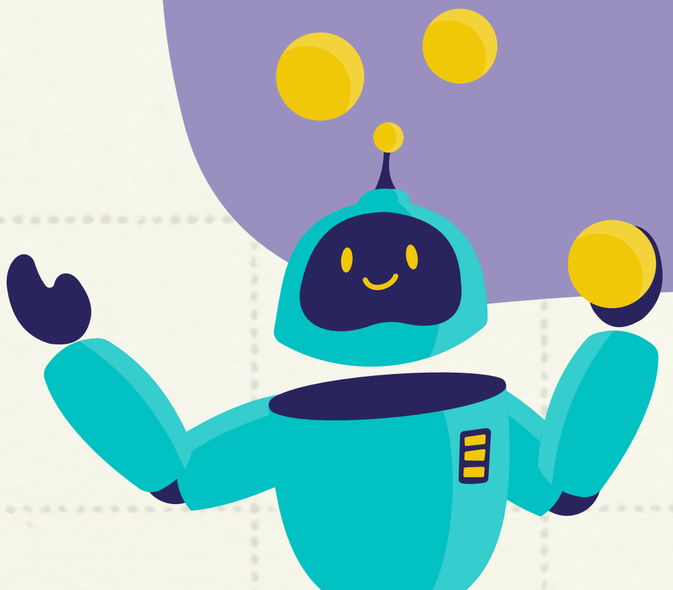


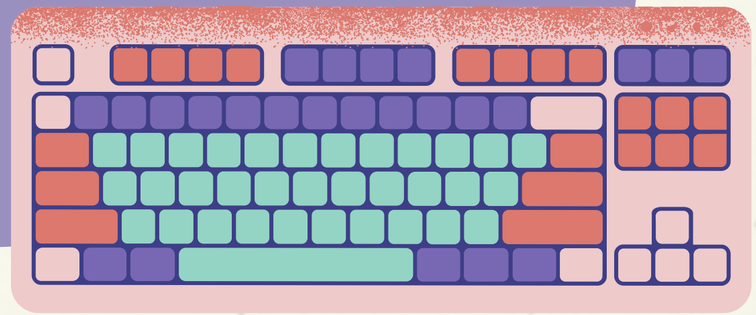


```python
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
print("Dictionary",my_dict)
my_dict['occupation'] = 'Engineer'
print("Dictionary",my_dict)
del my_dict['age']
print("Dictionary",my_dict)
keys = my_dict.keys()
print("Dictionary keys",keys)
values = my_dict.values()
print("Dictionary values",values)
age = my_dict.get('age')
print(" A certian Dictionary value ",age)
all_items = my_dict.items()
print("Dictionary all items",all_items)
city = my_dict.pop('city')
print("Dictionary",my_dict)
```

```
/Users/sanatwalia/.zprofile:1: not a directory: /opt/homebrew/bin/brew/shellenv
/usr/local/bin/python3 "/Users/sanatwalia/Documents/cpp learn /graphs/1.py"
(base) sanatwalia@Sanats-MacBook-Air cpp learn  % /usr/local/bin/python3 "/Users/san
atwalia/Documents/cpp learn /graphs/1.py"
Dictionary {'name': 'John', 'age': 30, 'city': 'New York'}
Dictionary {'name': 'John', 'age': 30, 'city': 'New York', 'occupation': 'Engineer'}
Dictionary {'name': 'John', 'city': 'New York', 'occupation': 'Engineer'}
Dictionary keys dict_keys(['name', 'city', 'occupation'])
Dictionary values dict_values(['John', 'New York', 'Engineer'])
 A certian Dictionary value  None
Dictionary all items dict_items([('name', 'John'), ('city', 'New York'), ('occupatio
n', 'Engineer')])
Dictionary {'name': 'John', 'occupation': 'Engineer'}
(base) sanatwalia@Sanats-MacBook-Air cpp learn  %
```

FUNCTIONS





- Functions are blocks of code that perform a specific task.

- They help in organizing code into reusable chunks.
- Defined using the def keyword.

- def greet(name):
  return f"Hello, {name}!"

- Call with greet("Alice") to get "Hello, Alice!".

- **Return Statement**
- Returns a value from the function.

- **Required Arguments:**

```
def add(x, y):
    return x + y
```

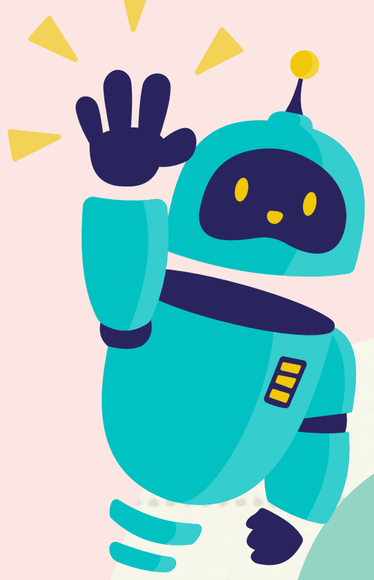
- Keyword Arguments:

```
def person_info(name, age):
    return f"Name: {name}, Age: {age}"
```

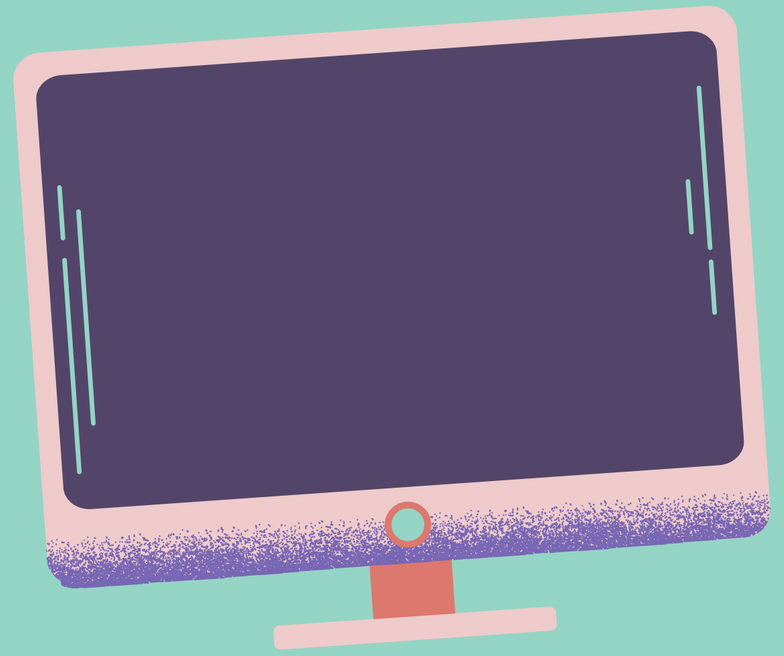
- Call with person_info(age=30, name="John")

- **Default Parameters:**Parameters with default values.

```
def greet(name="Guest"):
    return f"Hello, {name}!"
```



VARIABLE SCOPE

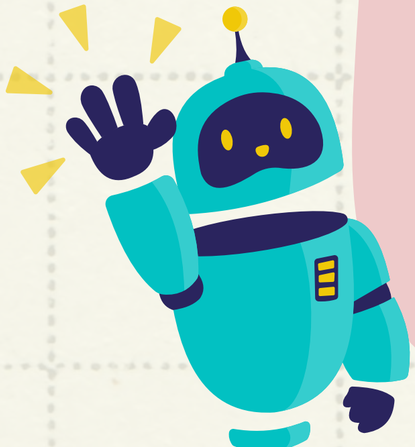


- **Global Variables:**
- Can be used outside the function

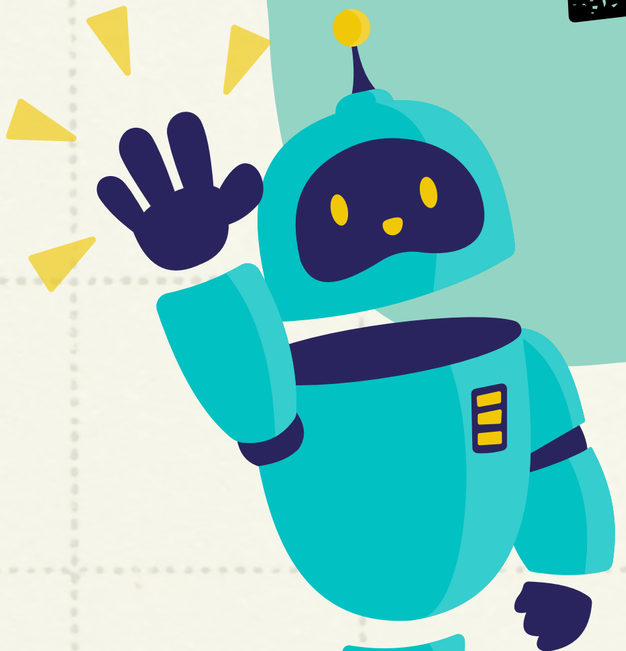
```
global_var = 10
def multiply(num):
        return global_var * num
```

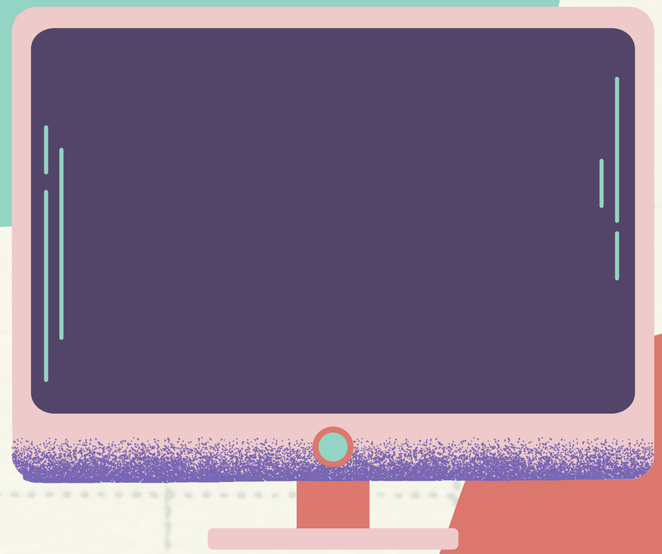
- Local Variables:
- Only Exist inside the function

```
def local_multiply(num):
        local_var = 5
        return local_var * num
```

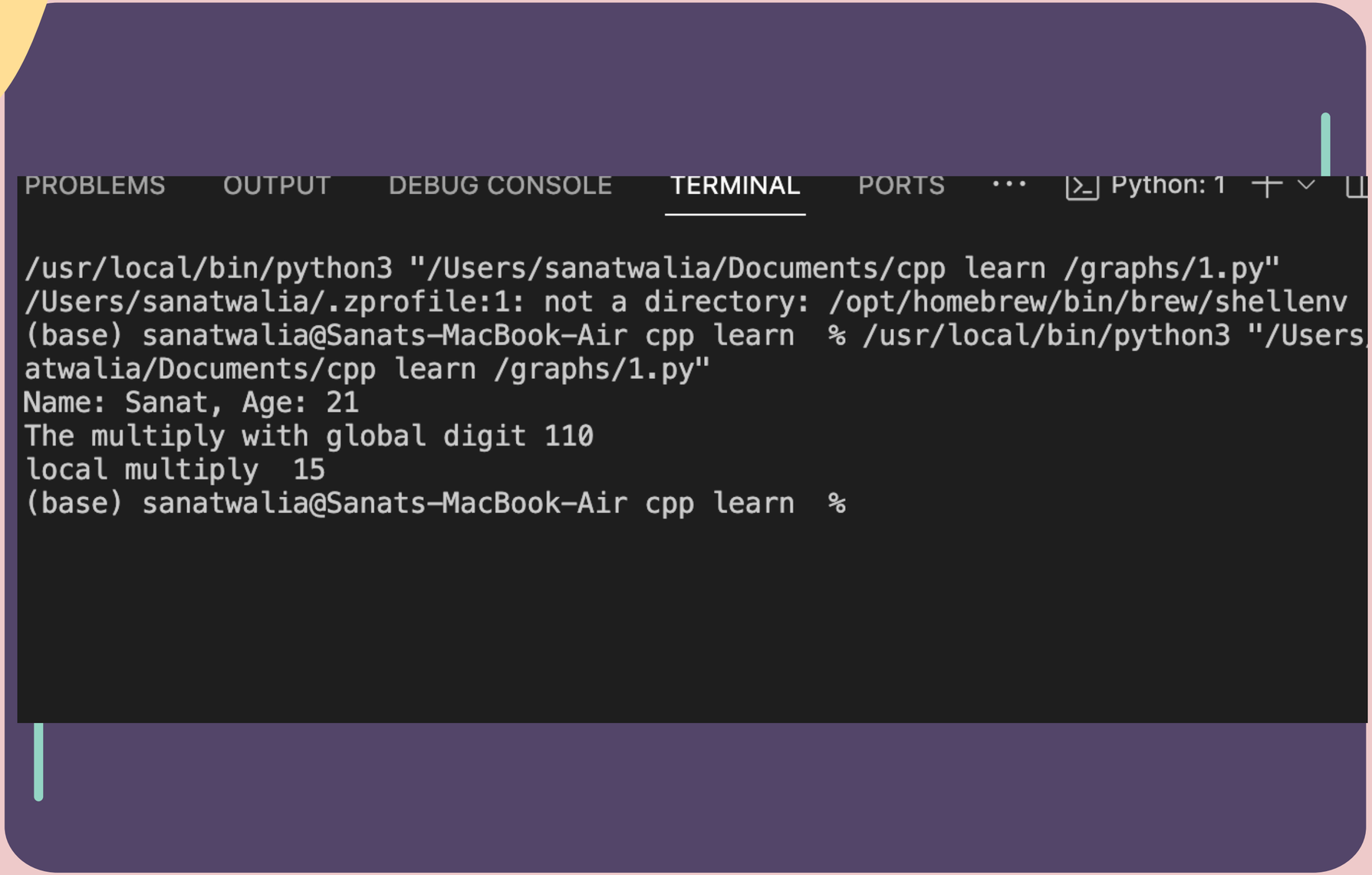


EXAMPLES

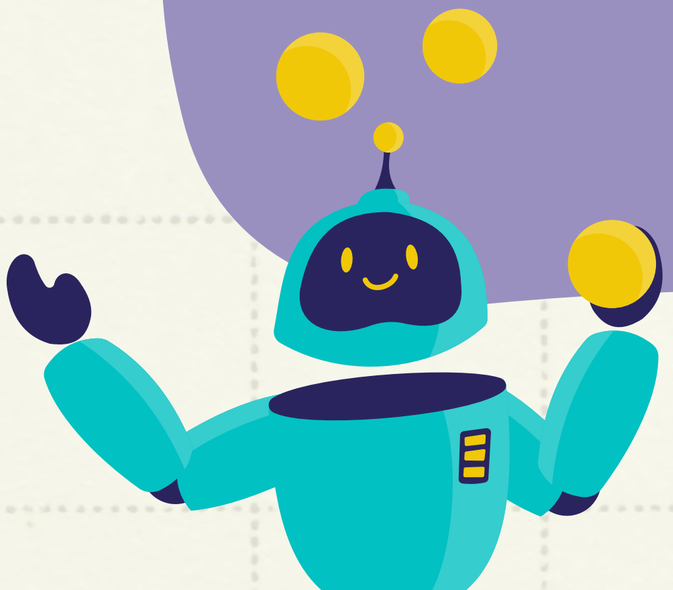


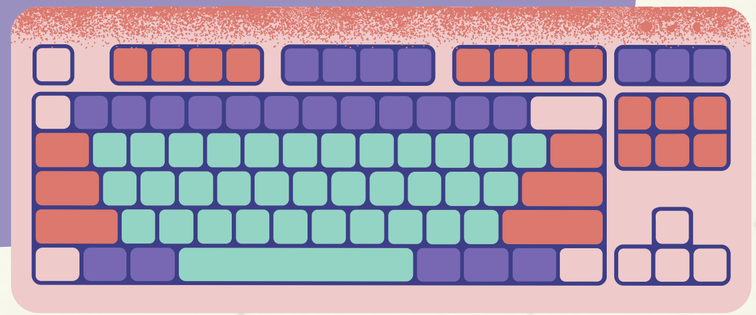


```
1
2   def person_info(name, age):
3       return f"Name: {name}, Age: {age}"
4   print(person_info('Sanat',21))
5   def greet(name="Guest"):
6       return f"Hello, {name}!"
7   global_var = 10
8   def multiply(num):
9       return global_var * num
10  print("The multiply with global digit",multiply(11))
11  def local_multiply(num):
12      local_var = 5
13      return local_var * num
14  print("local multiply ",local_multiply(3))
```


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Python: 1
/usr/local/bin/python3 "/Users/sanatwalia/Documents/cpp learn /graphs/1.py"
/Users/sanatwalia/.zprofile:1: not a directory: /opt/homebrew/bin/brew/shellenv
(base) sanatwalia@Sanats-MacBook-Air cpp learn % /usr/local/bin/python3 "/Users/
atwalia/Documents/cpp learn /graphs/1.py"
Name: Sanat, Age: 21
The multiply with global digit 110
local multiply 15
(base) sanatwalia@Sanats-MacBook-Air cpp learn %

CLASSES





- Python, as an object-oriented programming language, supports classes and objects
- Classes serve as blueprints for creating objects with similar attributes and methods
- user-defined data types that bundle data and functions together
- They encapsulate data for the purpose of efficient and easy management
- Class instances are individual objects created based on the class blueprint

- **Syntax:**

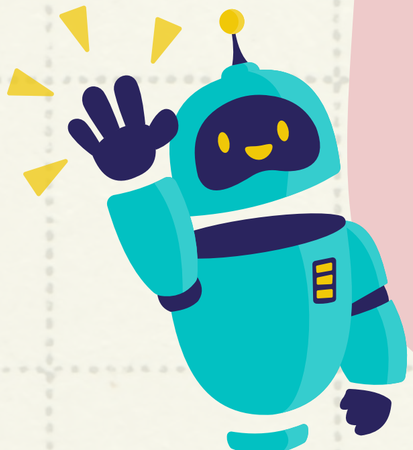
```
class ClassName:
    # Class attributes and methods definition
    def __init__(self, parameters):  # Constructor
        # Initialize instance variables
    def method_name(self, parameters):  # Method definition
        # Method body
```

- Constructor(init)

1. Definition:
   - The constructor, denoted by __init__, is a special method in Python classes.
   - It is automatically called when a new instance of the class is created.
2. Purpose:
   - The main purpose of the constructor is to initialize the object's state.
   - It is used to set initial values to the instance variables of the class.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Creating an instance of the Person class
person1 = Person("Alice", 30)
```



Instance Variables and Self

- Instance variables are unique to each object created from a class.
- They hold data that is specific to each instance of the class.

Understanding Self:

- self is a reference to the current instance of the class.
- It is the first parameter of any instance method in Python.
- Allows access to instance variables and methods within the class.

Declaring Instance Variables:

- Instance variables are declared within methods using self.
- They are initialized inside the constructor (__init__) or any other instance method.

Accessing Instance Variables:

- Instance variables are accessed using dot notation (self.variable_name).
- self refers to the current object, allowing access to its attributes.

```python
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

# Creating instances of the Car class
car1 = Car("Toyota", "Camry")
car2 = Car("Honda", "Accord")

# Accessing instance variables
print(car1.brand)  # Output: Toyota
print(car2.model)  # Output: Accord
```

encapsulation and abstraction

```python
# Encapsulation and Abstraction
class Vehicle:
    def __init__(self, name, fuel_type):
        self.__name = name  # Encapsulated attribute
        self.__fuel_type = fuel_type  # Encapsulated attribute

    def start(self):
        # Abstraction: Internal implementation details are hidden
        print(f"{self.__name} starts.")

    def refuel(self):
        # Abstraction: Internal implementation details are hidden
        print(f"Refueling {self.__fuel_type}...")
```

```python
# Creating instances
car = Car("Toyota Camry", "Petrol", "Automatic")
motorcycle = Motorcycle("Harley Davidson", "Gasoline", "V-Twin")

# Performing actions
perform_action(car)
perform_action(motorcycle)
```

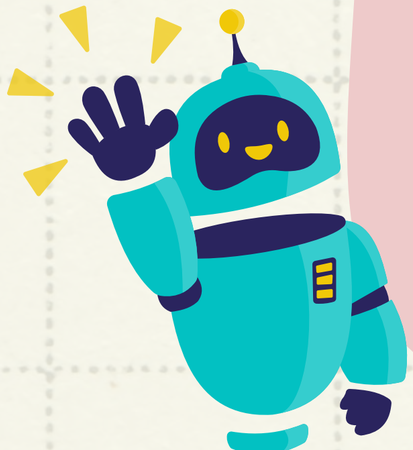
inheritance and polymorphism

```python
# Inheritance
class Car(Vehicle):
    def __init__(self, name, fuel_type, transmission):
        super().__init__(name, fuel_type)
        self.__transmission = transmission  # Encapsulated attribute

    def drive(self):
        print(f"{self._Vehicle__name} is driving with {self.__transmission} transmis


class Motorcycle(Vehicle):
    def __init__(self, name, fuel_type, engine_type):
        super().__init__(name, fuel_type)
        self.__engine_type = engine_type  # Encapsulated attribute

    def drive(self):
        print(f"{self._Vehicle__name} is riding with {self.__engine_type} engine.")
```



```python
# Creating instances
car = Car("Toyota Camry", "Petrol", "Automatic")
motorcycle = Motorcycle("Harley Davidson", "Gasoline", "V-Twin")

# Performing actions
perform_action(car)
perform_action(motorcycle)
```

EXAMPLE

```python
class Dog:
    # Class attribute
    species = "Canis familiaris"

    # Constructor (init method)
    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age    # Instance attribute

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

```python
# Creating instances of the Dog class
buddy = Dog("Buddy", 5)
miles = Dog("Miles", 4)

# Accessing instance attributes and calling instance methods
print(buddy.description())  # Output: Buddy is 5 years old
print(buddy.speak("Woof Woof!"))  # Output: Buddy says Woof Woof!

# Accessing class attribute
print(f"{buddy.name} is a {buddy.species}")  # Output: Buddy is a Canis familiaris

# Modifying instance attributes
miles.age = 6
print(miles.description())  # Output: Miles is 6 years old
```