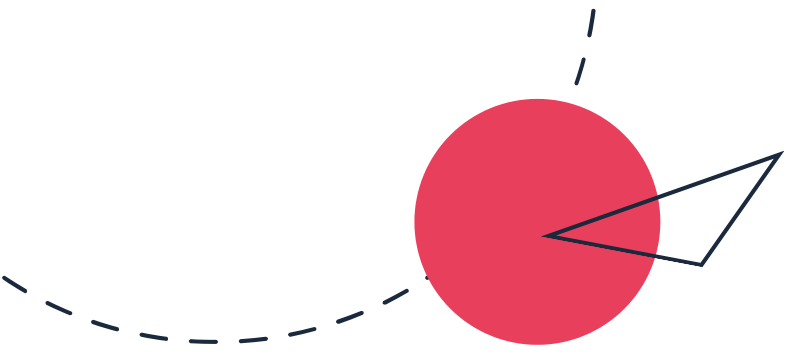




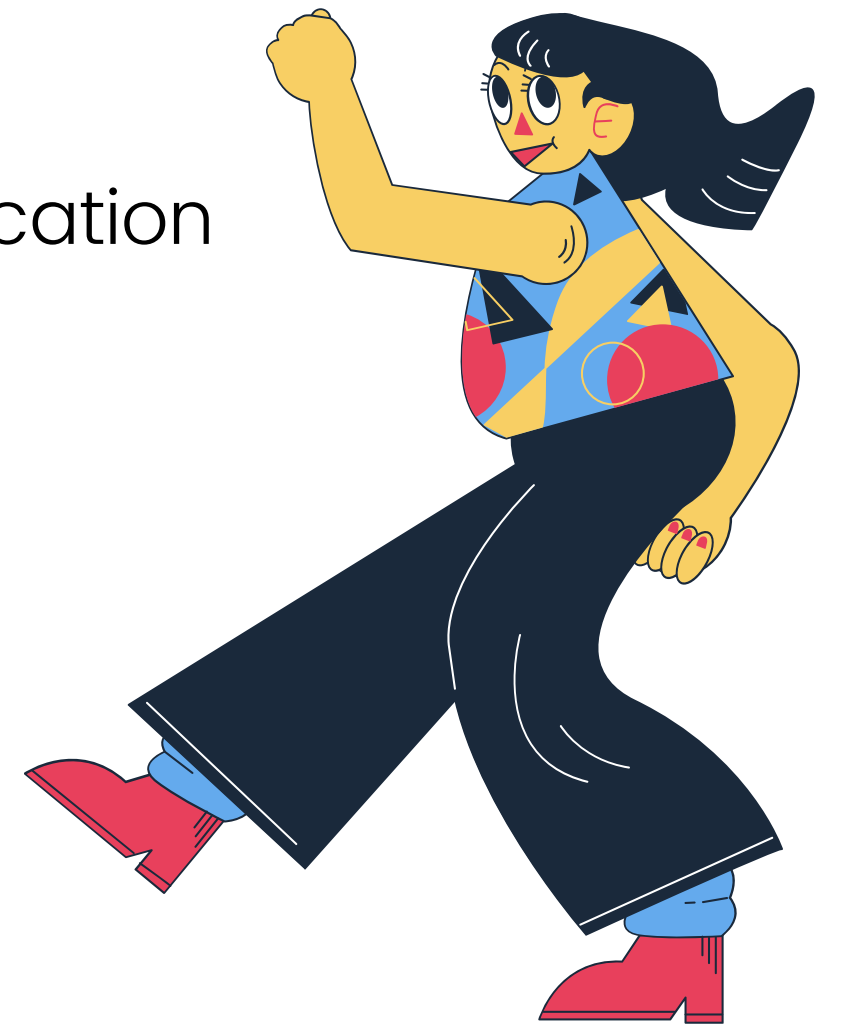
FLOXUS EDUCATION

# Python Bootcamp



# Agenda

- Python setup
- Variables
- Python Operators
- Control Statements
- String, Number
- List, Tuple, Dictionary, and Sets
- Working with Sequences
- Mini Project
- Exceptions and Error Trapping
- Class, Object and Functions
- Modules
- Working with Files
- Packages in Python
- Python-Database Communication
- Multi-Threading
- Unit Testing with PyTest
- Python Web Frameworks
- GUI -Tkinter
- Final Project.

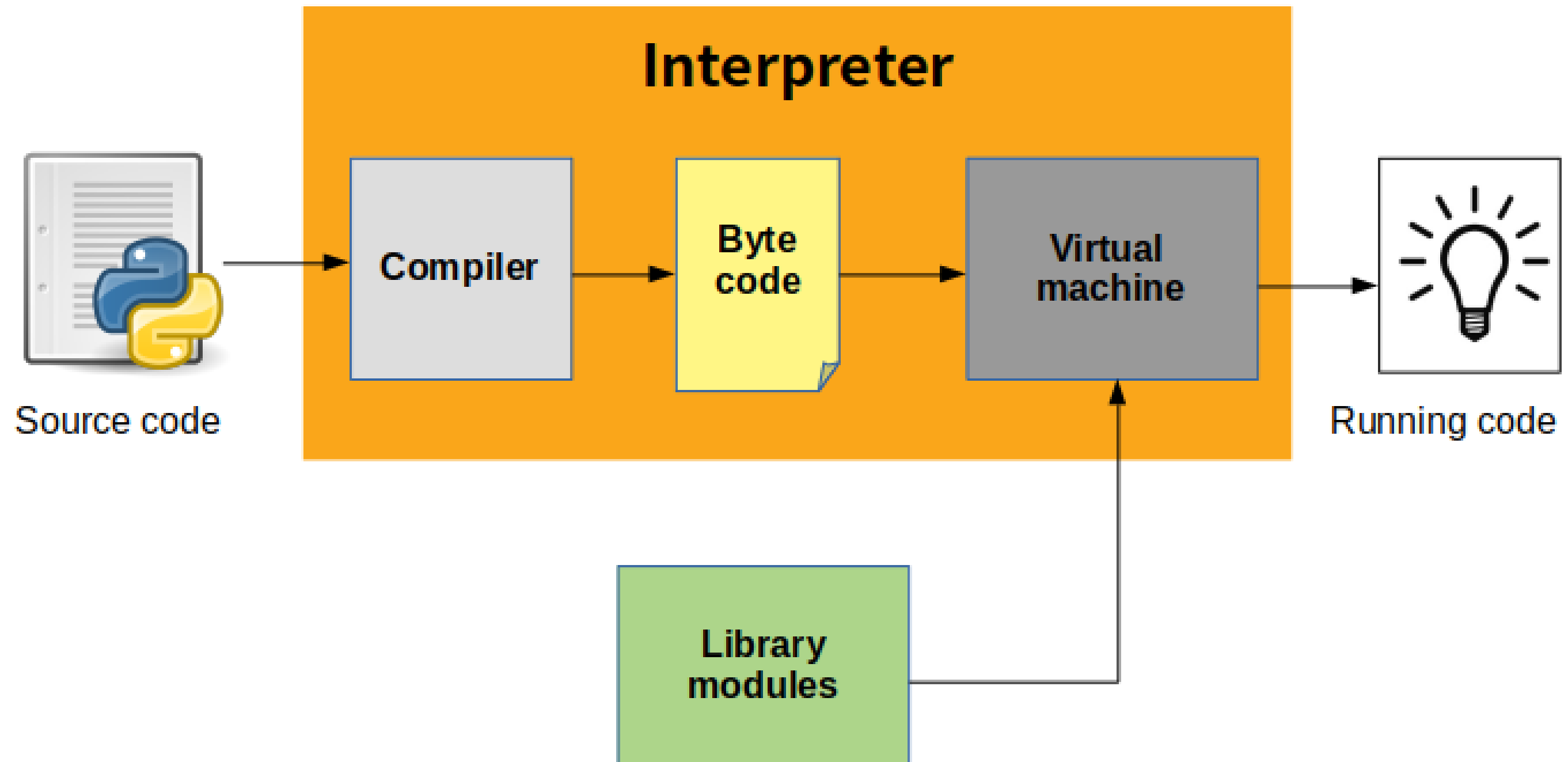


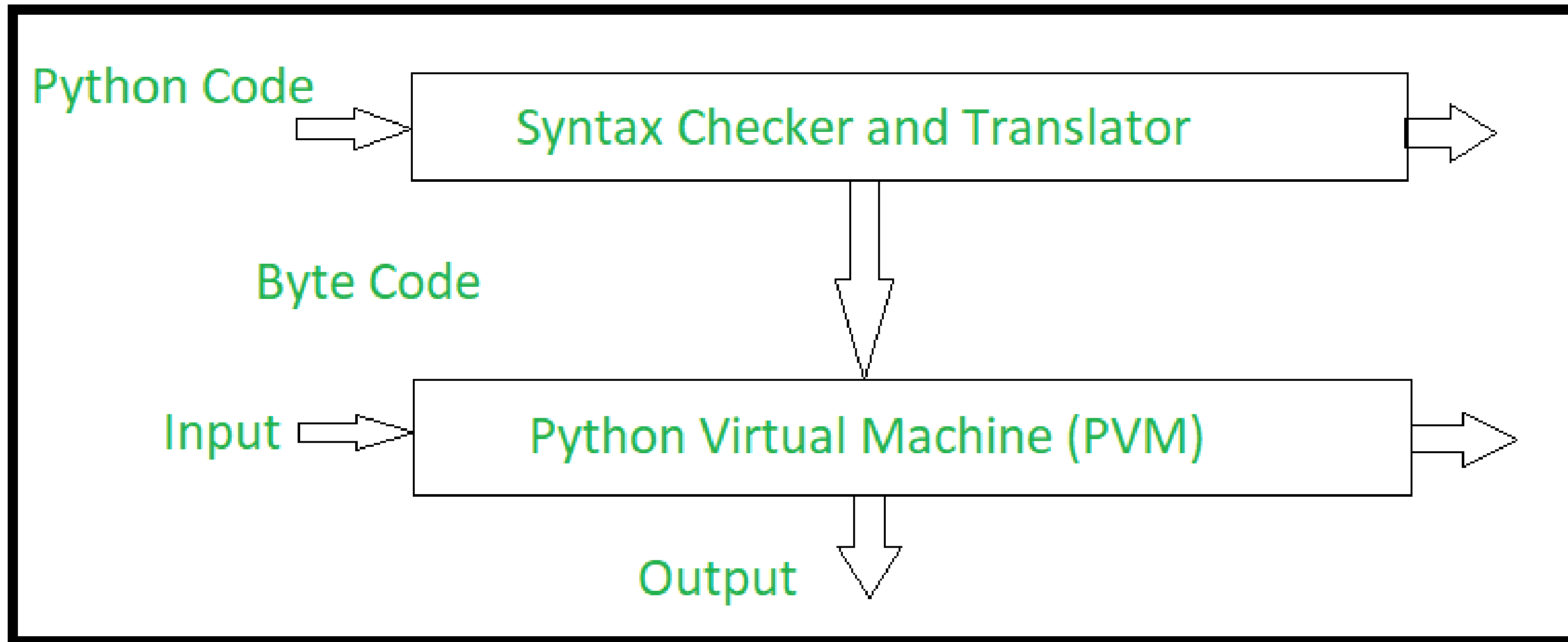
# What is Python

Python is a high-level interpreted language that employs an object-oriented approach.

Python is more preferred because of its simplicity, powerful libraries, and readability.

Python runs on an interpreter system, meaning that code can be executed as soon as it is written.





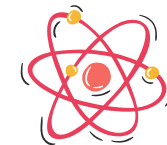
# APPLICATIONS OF PYTHON



WEB TECHNOLOGIES



GAME DEVELOPMENT



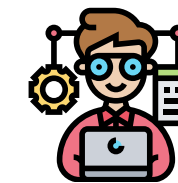
SCIENTIFIC AND NUMERIC APPLICATIONS



ENTERPRISE-LEVEL/BUSINESS APPLICATIONS

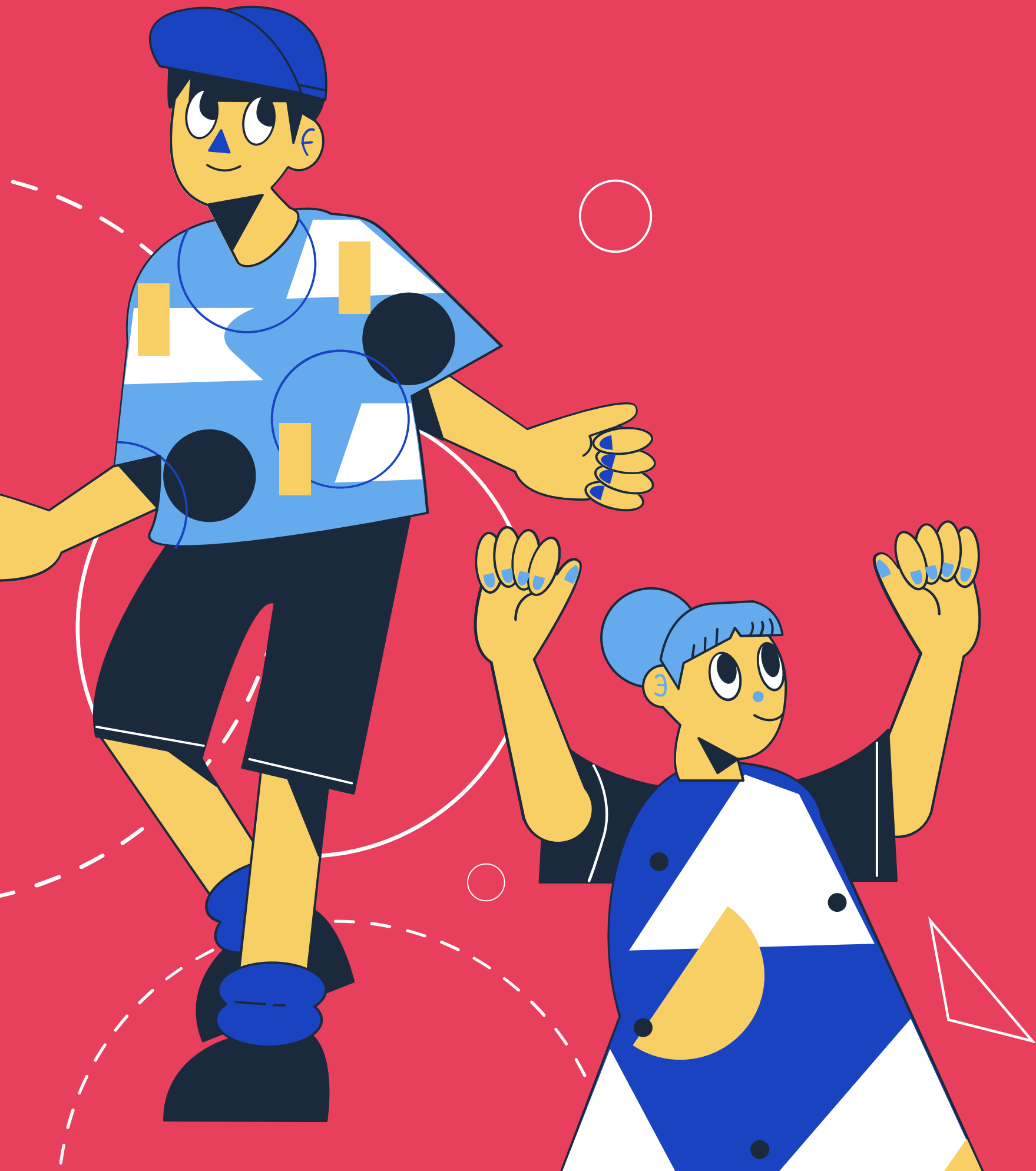


ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING



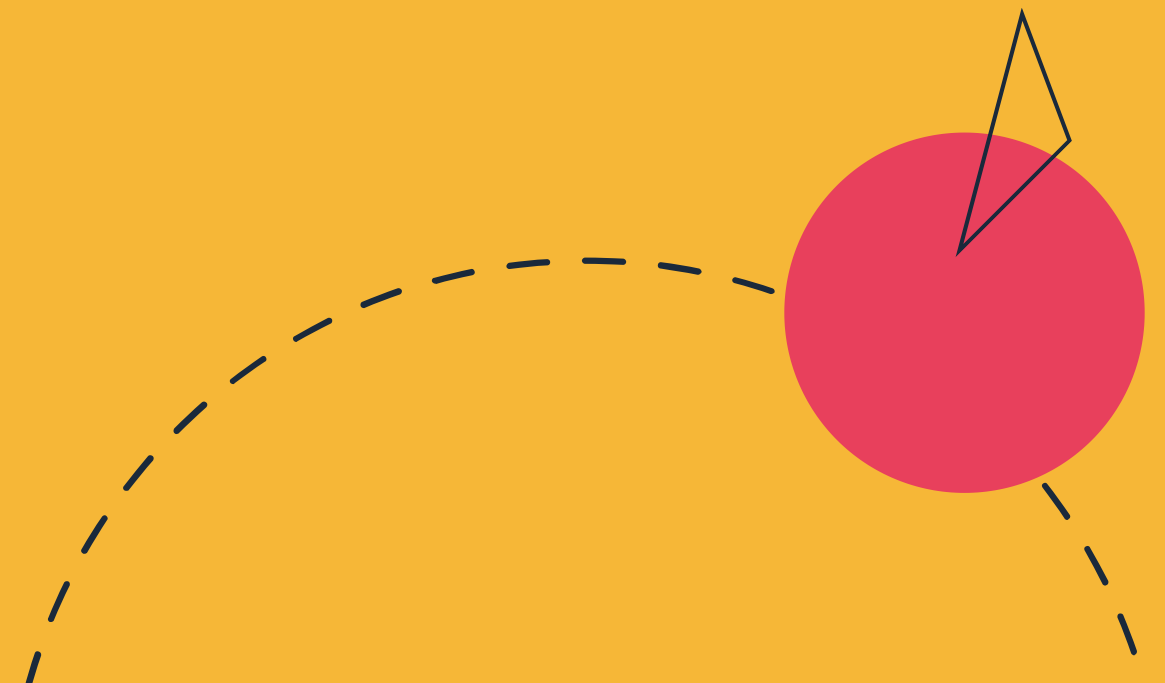
SOFTWARE DEVELOPMENT

MANY MORE..



# Install Python

# Variables



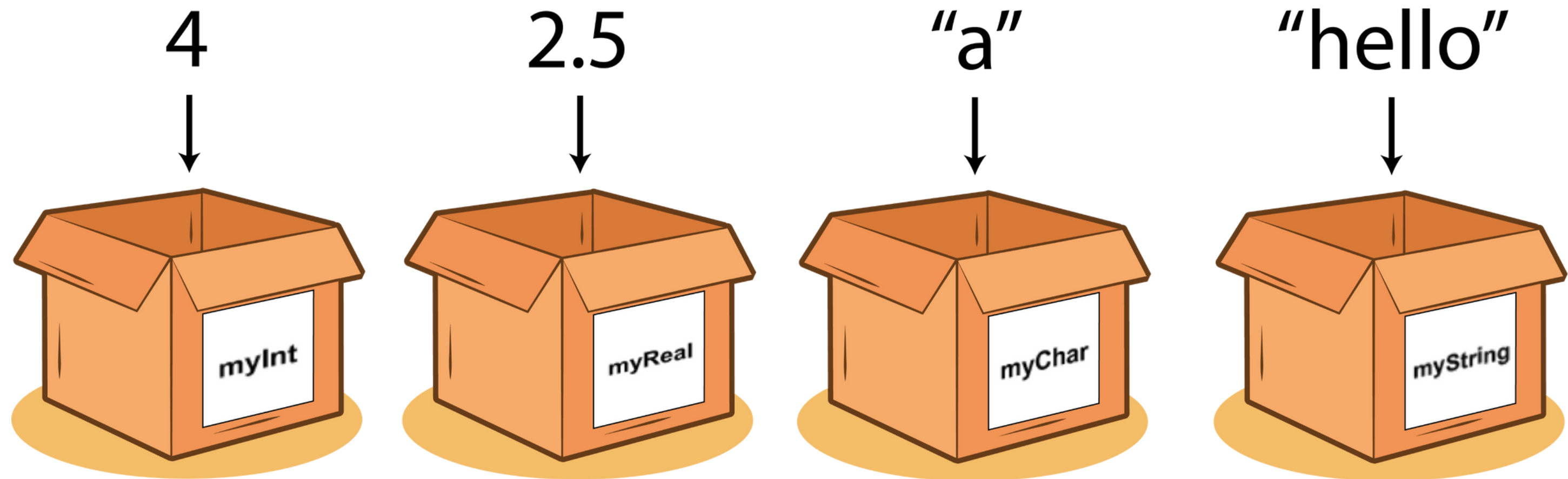


# Variables

Variables are containers for storing data values.

Syntax

```
x = 5  
y = "Floxus"  
print(x)  
print(y)
```



Variables with different data types

**OPEN IT**



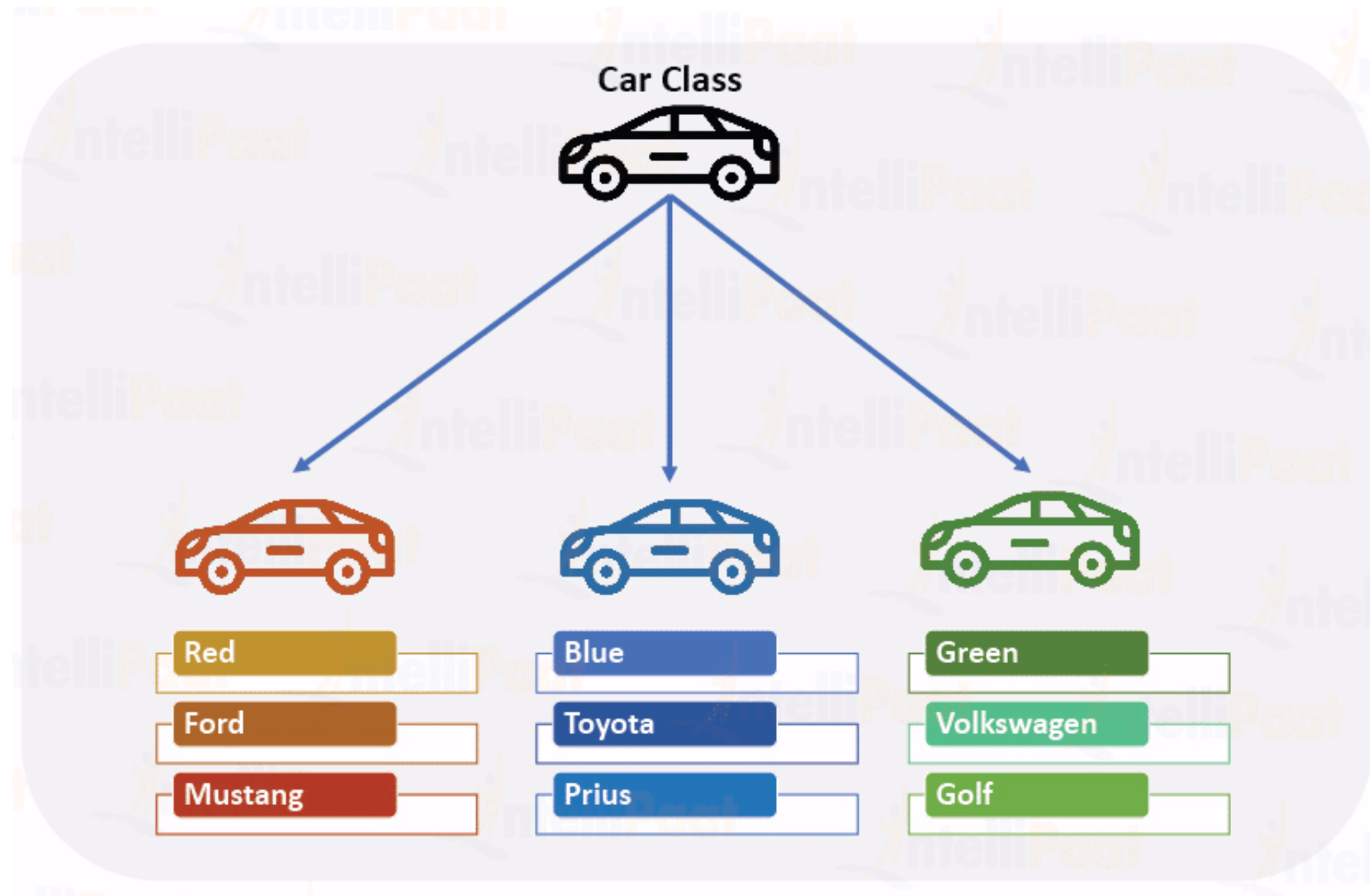
- Variables do not need to be declared with any particular type, and can even change type after they have been set (type casting). It works as placeholder.
- Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes. There are various data types in Python.

Following are the standard or built-in data type of Python:

- Numeric (Integer, Complex Number, Float)
- Sequence (String, list, Tuple)
- Boolean (True, False)
- Set
- Dictionary

Let's Code it 

# Class and Objects

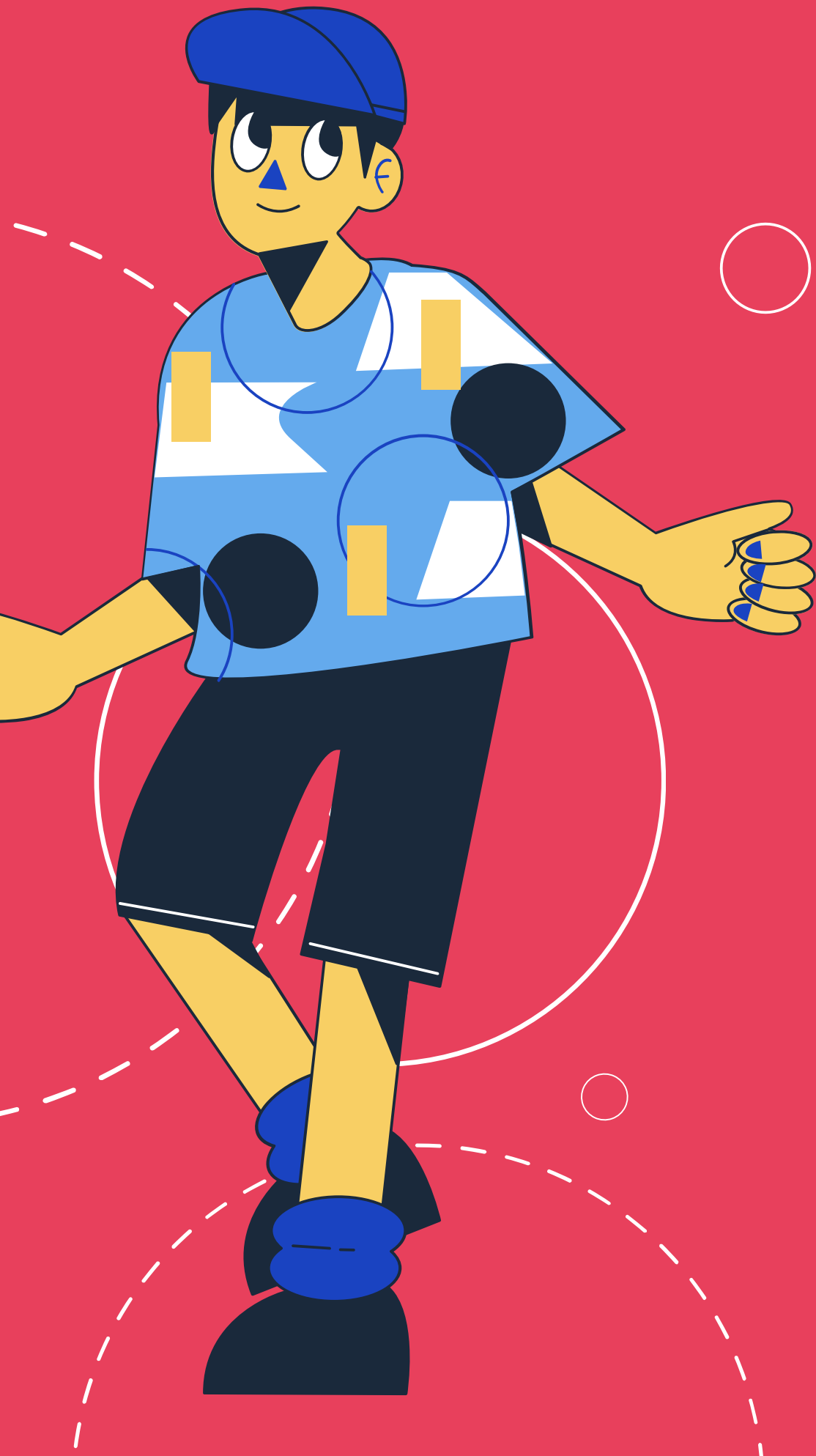


Let's Code it →



**Thank You!**



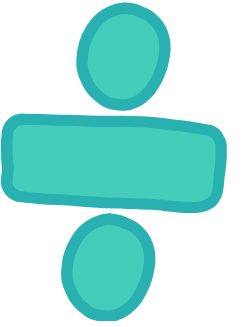


# Python Operators





Operators are used to perform operations on variables and values.



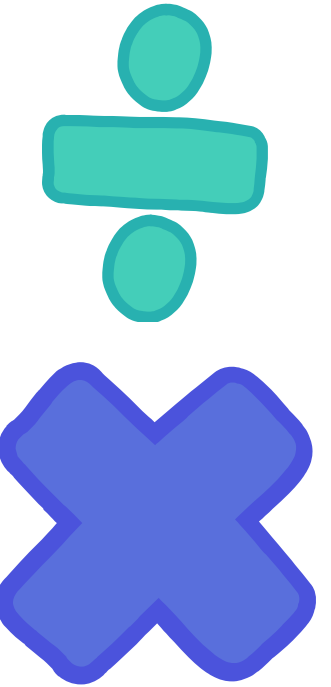
Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators



# Arithmetic Operators

- $+$  (Addition  $x + y$ )
- $-$  (Subtraction  $x - y$ )
- $*$  (Multiplication  $x * y$ )
- $/$  (Division  $x / y$ )
- $\%$  (Modulus  $x \% y$ )
- $**$  (Exponentiation  $x ** y$ )
- $//$  (Floor division  $x // y$ )



Let's Code it 

# Assignment Operators

Operator	Example	Equals To
=	a = 10	a = 10
+=	a += 10	a = a+10
-=	a -= 10	a = a-10
*=	a *= 10	a = a*10
/=	a /= 10	a = a / 10
%=	a %= 10	a = a % 10
//=	a //= 10	a = a // 10
**=	a **= 10	a = a ** 10
&=	a &= 10	a = a & 10
=	a  = 10	a = a  10
^=	a ^= 10	a = a ^10
>>=	a >>= 10	a = a >> 10
<<=	a <<= 10	a = a << 10

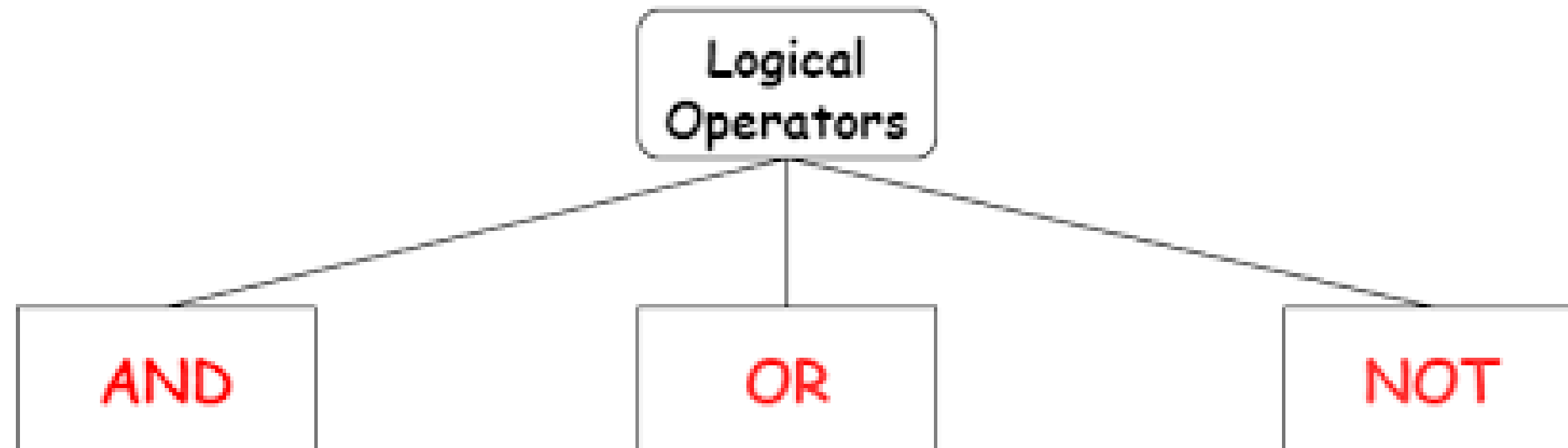
Let's Code it →

# Comparison Operators

Operator	Meaning
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code>&gt;</code>	Greater than
<code>&lt;</code>	Less than
<code>&gt;=</code>	Greater than or equal to
<code>&lt;=</code>	Less than or equal to

Let's Code it 

# Logical Operators



$x < 5$  and  $x < 10$

$x < 5$  or  $x < 4$

$\text{not}(x < 5 \text{ and } x < 10)$

Let's Code it 

# Identity Operators

Operator	Description
is	It returns true if two variables point the same object and false otherwise
is not	It returns false if two variables point the same object and true otherwise

x is y

x is not y

Let's Code it 

# Membership Operators

Operator	Description
in	It returns true if value/variable is found in the sequence and false otherwise
not in	It returns true if value/variable is not found in the sequence and false otherwise

x in y

x not in y

Let's Code it →

# Bitwise Operators

Bitwise operators are used to compare (binary) numbers

Operator	Meaning
<b>&amp;</b>	Bitwise AND
<b> </b>	Bitwise OR
<b>^</b>	Bitwise exclusive OR / Bitwise XOR
<b>~</b>	Bitwise inversion (one's complement)
<b>&lt;&lt;</b>	Shifts the bits to left / Bitwise Left Shift
<b>&gt;&gt;</b>	Shifts the bits to right / Bitwise Right Shift

Let's Code it 



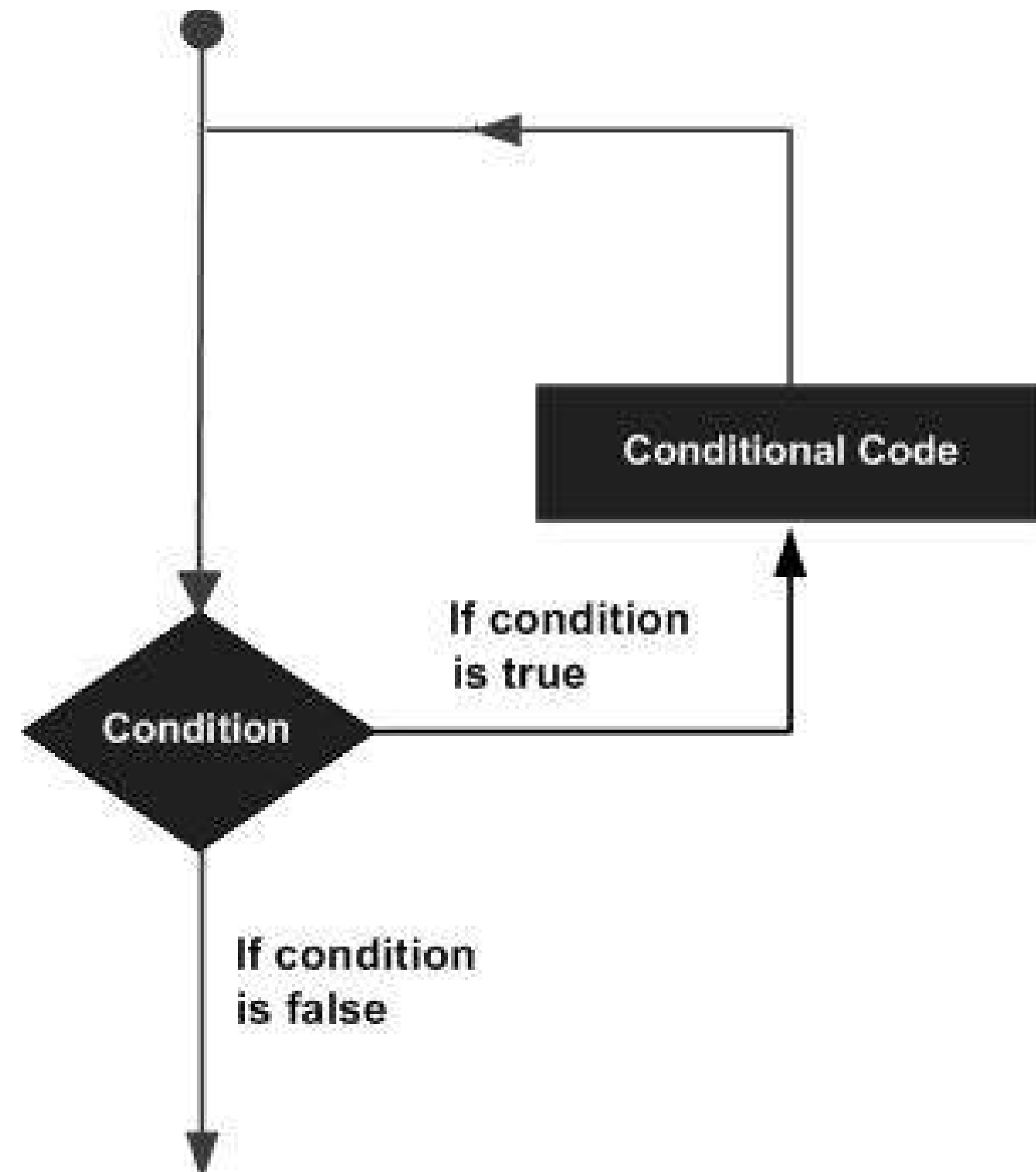
A decorative graphic on the left side of the slide. It features a solid red circle, a dashed black line that curves upwards and to the right, and a solid blue circle. The red circle is positioned on the left side of the slide, and the blue circle is positioned on the right side. The dashed line starts from the bottom left and curves upwards and to the right, passing through the red circle.

**Thank You**



# Loops & Control Statements

A loop statement allows us to execute a statement or group of statements multiple times.



# Loop Type

- **while loop** – Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
- **for loop** – Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
- **nested loop** – You can use one or more loop inside any other while, for or do..while loop.

# Syntax

## while loop

while expression:  
statement

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

Let's Code it →

# Syntax

## for in loop:

for iterating\_var in sequence:  
statements



```
list = [1, 2, 3]  
for x in list:  
    print(x)
```

Let's Code it →

# Syntax

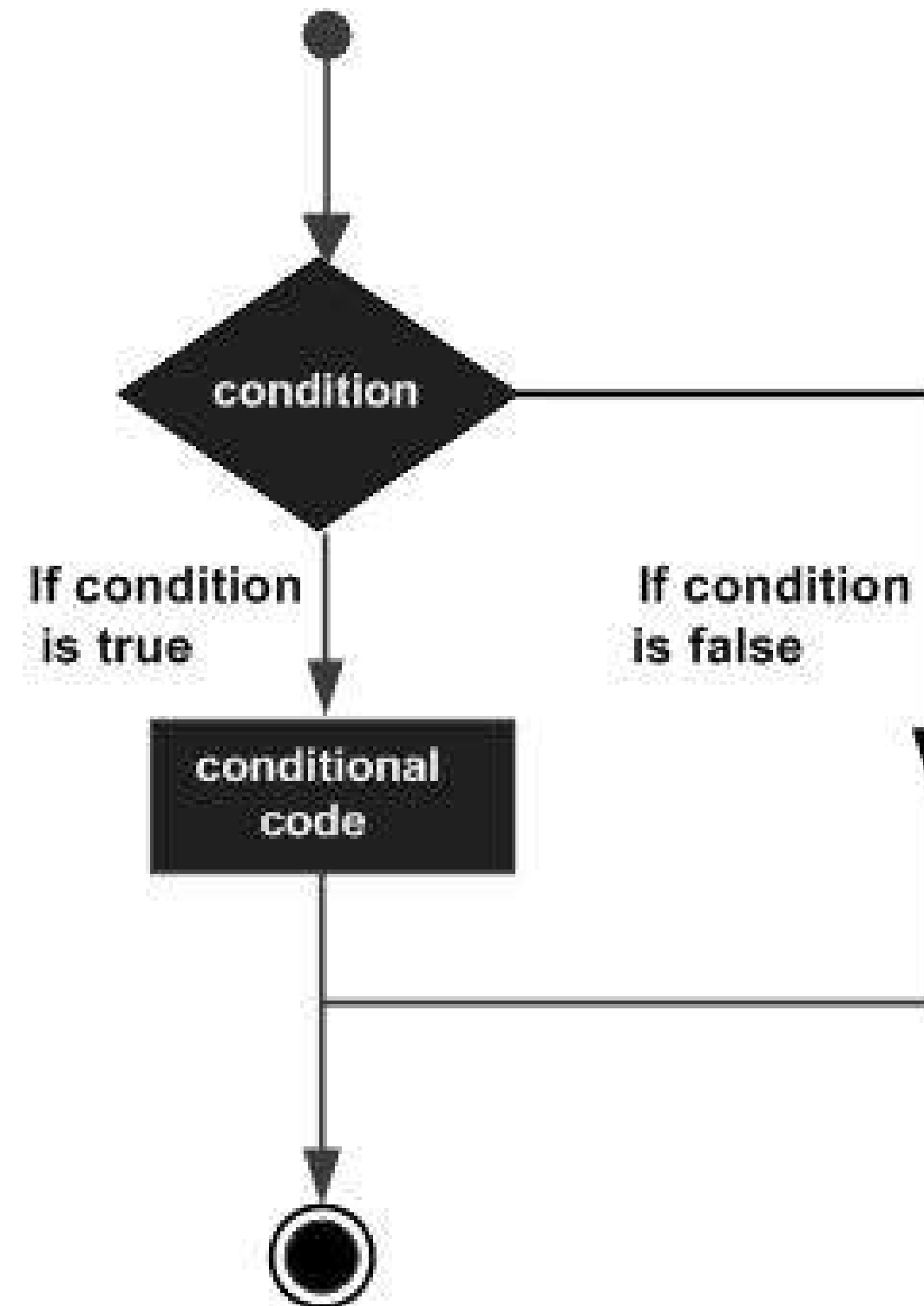
## Nested loop:

```
for iterating_var in sequence:
    for iterating_var in sequence:
        statements
        statements
```

```
while expression:
    while expression:
        statement
        statement
```

Let's Code it →

# Decision Making





# Conditional Statements

- **if statement**– An if statement consists of a boolean expression followed by one or more statements.
- **if-else statement**– An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE.
- **nested if-else statement**– You can use one if or else if statement inside another if or else if statement(s).

# Conditional Statements

## Syntax

```
if <expr>:  
    <statement>
```

```
if <expr>:  
    <statement(s)>  
else:  
    <statement(s)>
```

```
if <expr>:  
    <statement(s)>  
elif <expr>:  
    <statement(s)>  
elif <expr>:  
    <statement(s)>
```

Let's Code it →

# Conditional Statements

## Examples:

```
• • •  
  
x = 0  
y = 5  
if x < y:  
  
    print('yes')
```

```
• • •  
  
x = 1  
if x == 1:  
    print('Hello')  
elif x == 2:  
    print('Hiii')
```

Let's Code it →

# Control Statements

Control statements in python are used to control the order of execution of the program based on the values and logic

- **Continue**– When the program encounters a continue statement, it will skip the statements which are present after the continue statement inside the loop and proceed with the next iterations
- **Break**– The break statement is used to terminate the loop containing it, the control of the program will come out of that loop.
- **Pass** – We use a pass statement to write empty loops. Pass is also used for empty control statement, functions, and classes.

# Control Statements:

```
for char in 'Python':  
    if (char == 'y'):  
        continue  
    print(char)
```

```
for char in 'Python':  
    if (char == 'h'):  
        break  
    print(char)
```

```
for char in 'Python':  
    if (char == 'h'):  
        pass  
    print(char)
```

Let's Code it →



**Thank You!**



# Strings & Numbers

Strings in python are surrounded by either single quotation marks, or double quotation marks.

```
a = "Hello"  
b = 'Hello'  
print(a)
```

for multiline string use three quotes to assign to a variable




- Square brackets can be used to access elements of the string.



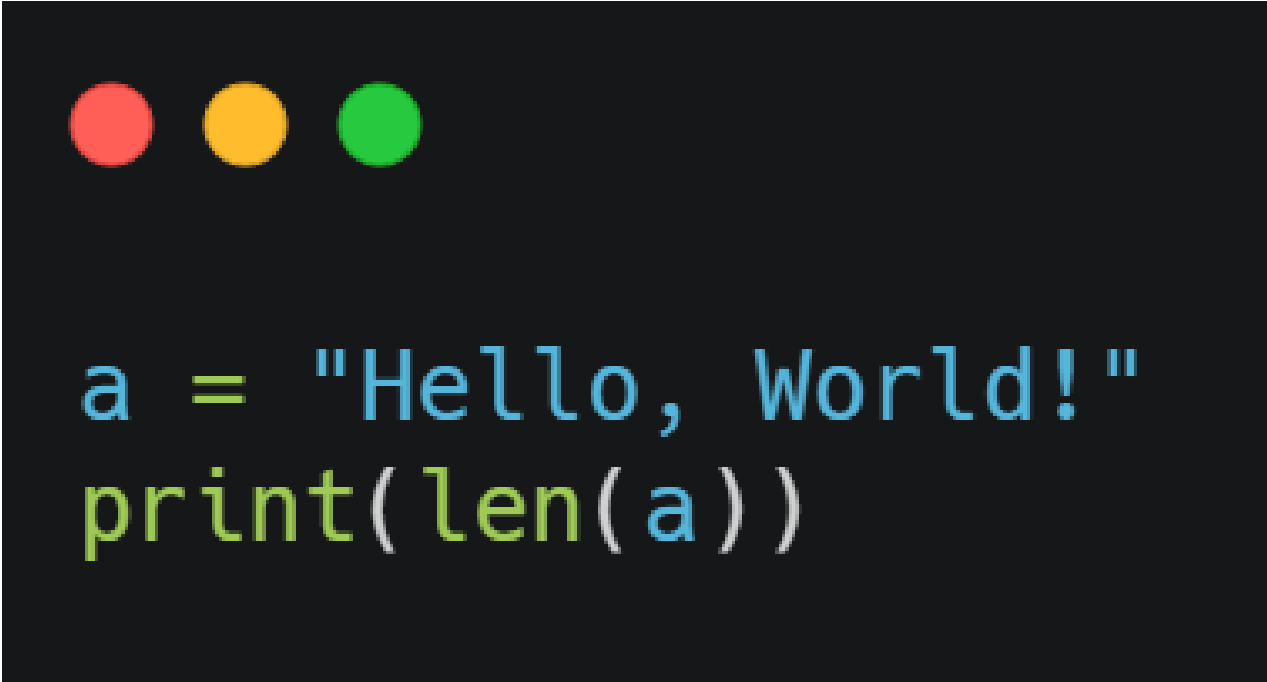
```
a = "Hello, World!"  
print(a[1])
```

- Strings in python works like an array



```
for x in "Floxus":  
    print(x)
```

- To get the length of a string, use the `len()` function

A dark-themed terminal window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains two lines of Python code: `a = "Hello, World!"` and `print(len(a))`.

```
a = "Hello, World!"  
print(len(a))
```



String[range]

Slicing

String[range] + 'abc'

Updating

String1 + String2

Concatenation

Repetition

String1 \* n

Membership

In, not in

Reverse

String[::-1]



# Numbers

---

There are three numeric types in Python:

- int
- float
- complex



```
x = 1      # int
y = 2.8    # float
z = 2+1j    # complex
```

# Type Conversion

---

You can convert from one type to another with the `int()`, `float()`, and `complex()` methods



```
x = int(1)    # x will be 1
y = int(2.8)  # y will be 2
z = int("3")  # z will be 3
```



```
x = 1        # int
y = 2.8      # float
z = 1j       # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```



THANK  
YOU



# Sequences

List, Tuple, Dictionary, Sets



# List

- Lists are used to store multiple items in a single variable.
- A list is a compound data type, you can group values together
- A list contains items separated by commas and enclosed within square brackets [ ].
- Items belonging to a list can be of different data types.
- The values stored in a list can be accessed using the [ ] operator
- Index starts at 0.
- List items are ordered, changeable, and allow duplicate values.



```
empty_list = []  
mylist = ["apple", "banana", "cherry"]  
mixed_type_list = [1, "banana", 1.5, True]
```

Append

→ `List.append(item)`

Insert

→ `List.insert(index, item)`

Extend

→ `List.extend(list1)`

Index

→ `List.index(item)`

Remove

→ `List.remove(item)`

Sort

→ `List.sort()`

Reverse

→ `List.reverse()`

# Loop Lists



```
thislist = ["apple", "banana", "cherry"]  
for x in thislist:  
    print(x)
```

```
i = 0  
while i < len(thislist):  
    print(thislist[i])  
    i = i + 1
```

# Tuples

- Tuples are used to store multiple items in a single variable.
- A tuple is a collection that is ordered and unchangeable(immutable).
- Tuples are written with round brackets.
- The parentheses are optional but is a good practice to write it.
- A tuple can have any number of items and they may be of different types (integer, float, list, string etc.).
- Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is in fact a tuple
- If the element is itself a mutable datatype like list, its nested items can be changed.
- We cannot delete or remove items from a tuple. But deleting a tuple entirely is possible using the keyword del.



```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d"
tup = ()
tup4= (50,)
print "tup1[0]: ", tup1[0];
print "tup2[1:5]: ", tup2[1:5];
```

Index

→ `Tuple.index(item)`

Slicing

→ `Tuple[range]`

Concatenation

→ `Tuple1+Tuple2`

Repetition

→ `Tuple*n`

Count

→ `Tuple.count(item)`

# Dictionary

- As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.
- Each element in a dictionary is represented by a key: value pair.
- Dictionaries are optimized to retrieve values when the key is known.
- Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
- Creating a dictionary is as simple as placing items inside curly braces `{}` separated by a comma.
- While values can be of any data type and can repeat, keys must be of immutable type and must be unique
- Dictionary are mutable. We can add new items or change the value of existing items using assignment operator.
- If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.





```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict["brand"])  
x = thisdict.keys()  
  
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
x = car.keys()  
  
print(x) #before the change  
  
car["color"] = "white"  
  
print(x) #after the change
```

Length



`len(dict)`

Delete



`dict.pop(key)`

Membership



`in, not in`

Change Items



`dict.update({key: value})`

# Set

- A set is an unordered collection of items. Every element is unique and must be immutable.
- However, the set itself is mutable. We can add or remove items from it.
- Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.
- A set is created by placing all the elements inside curly braces {}, separated by comma or by using the built-in function set().
- The elements can be of different types (integer, float, tuple, string etc.).
- Sets are unchangeable, meaning that we cannot change the items after the set has been created.
- Sets cannot have two items with the same value.



```
myset = {"apple", "banana", "cherry"}  
print(thisset)  
set1 = {"apple", "banana", "cherry"}  
set2 = {1, 5, 7, 9, 3}  
set3 = {True, False, False}  
set1 = {"abc", 34, True, 40, "male"}  
thisset = set(("apple", "banana", "cherry"))
```

Add Items

Clear

Copy

Difference

Discard

Remove

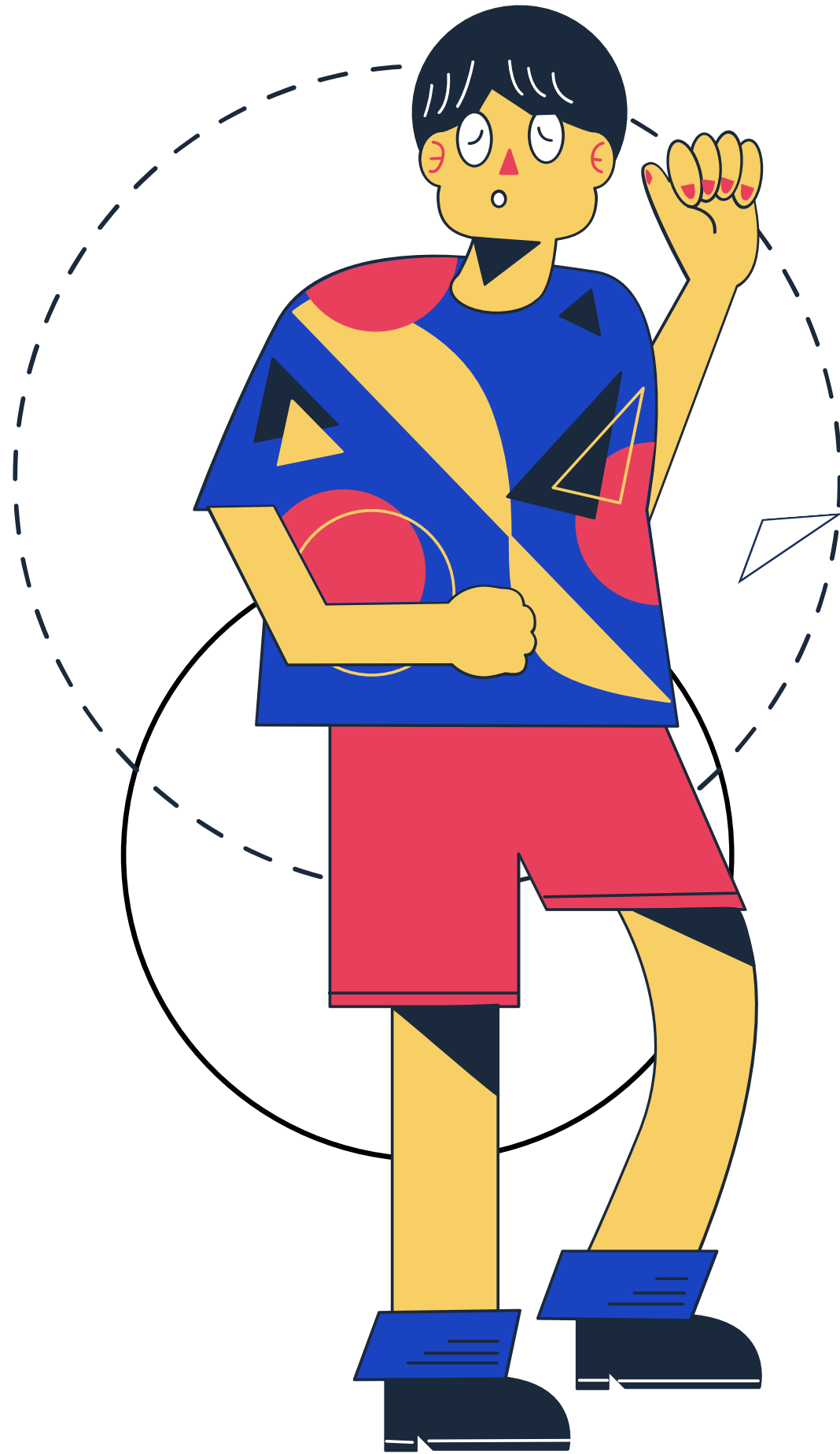
Intersection

# Summary

- List is a collection which is ordered and changeable. Allows duplicate members.
- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- Set is a collection which is unordered and unindexed. No duplicate members.
- Dictionary is a collection which is ordered and changeable. No duplicate members

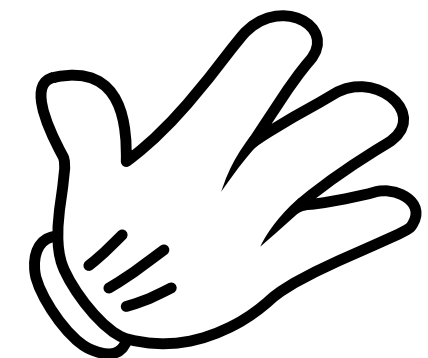
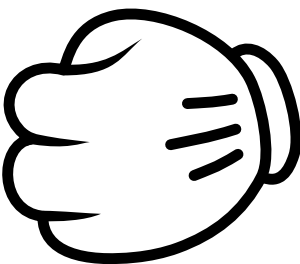
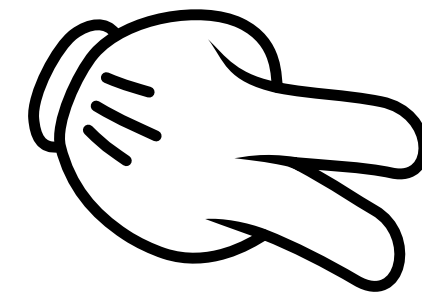


THANK  
YOU



# Mini Project

Rock, Paper and Scissors Game





# Functions



# FUNCTIONS

---

- A function is a block of code that only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.



```
def my_function():  
    print("Hello from a function")  
  
my_function()
```



# ARGUMENTS

---

**Parameters or Arguments? The terms parameter and argument can be used for the same thing: information that are passed into a function.**

- A parameter is the variable listed inside the parentheses in the function definition.
- An argument is a value that is sent to the function when it is called.
- A function must be called with the correct number of arguments.
- If you do not know how many arguments that will be passed into your function, add a \* before the parameter name in the function definition.





```
def my_function(Carname):  
    print(fname + "Ford")
```

```
my_function("Honda")  
my_function("Toyota")  
my_function("BMW")
```



```
def my_function(*Cars):  
    print("Brand Name " + Cars[2])
```


```
my_function("Ford", "BMW", "Toyota")
```



# KEYWORD ARGUMENTS

---

Keyword arguments (or named arguments) are values that, when passed into a function, are identifiable by specific parameter names.



```
def my_function(child3, child2, child1):  
    print("The youngest child is " + child3)  
  
my_function(child1 = "Alice", child2 = "Ani", child3 = "Lily")
```



# DEFAULT PARAMETER VALUE

---



```
def my_function(country = "Norway"):  
    print("I am from " + country)
```

```
my_function("Sweden")  
my_function("India")  
my_function()  
my_function("Brazil")
```



# RECURSION

---

- Python also accepts function recursion, which means a defined function can call itself.
- Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.
- The developer should be very careful with recursion as it can be quite easy to slip into writing a function that never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming





```
def tri_recursion(k):  
    if(k > 0):  
        result = k + tri_recursion(k - 1)  
        print(result)  
    else:  
        result = 0  
    return result  
  
print("\n\nRecursion Example Results")  
tri_recursion(6)
```





# LGB RULE

---

Local, Global and Built-in





```
radius=2  
pi=3.141592654  
def area(radius):  
    area=pi*(pow(radius,2))  
    return area  
  
print area(4)
```




# LAMBDA

---

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.

lambda arguments : expression




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

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

# PASS BY OBJECT REFERENCE

---


- Python supports call by value (Where the value is always an object reference, not the value of the object)



```
def add(list):  
    list.append("JAVA")
```

```
list=["Python"]  
add(list)  
print(list)
```

Output:  
["Python", "Java"]



```
def add(list):  
    list=["JAVA"]
```

```
list=["Python"]  
add(list)  
print(list)
```

Output:  
["Python"]



# Classes and Objects



# Object-Oriented Programming

---

- Object-Oriented Programming (OOP) is an approach of looking at a problem using models which are organized around the real-world concepts.
- The fundamental construct of Object-Oriented Programming (OOP) is object which combines both data structure and behavior as a single entity.

Features of object-oriented programming are:-

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

# CLASSES AND OBJECTS

---

- A class is simply a logical entity that behaves as a prototype or a template to create objects, while an object is just a collection of variables and Python functions.
- Variables and functions are defined inside the class and are accessed using objects.
- These variables and functions are collectively known as attributes.
- Python is an object-oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor or a "blueprint" for creating objects.
- To create a class, use the keyword class



# Public, Protected and Private Attributes

---

- Variables with the public access modifiers can be accessed anywhere inside or outside the class
- the private variables can only be accessed inside the class
- while protected variables can be accessed within the same package.

name - Public  
\_name-Protected  
\_\_name-Private





```
class MyClass:  
    x = 5  
p1 = MyClass()  
print(p1.x)
```





```
class Floxus:  
a = 5  
def function1(self):  
print('Welcome to Floxus')#creating a new object named object1 using class object  
object1 = Floxus()  
  
#accessing the attributes using new object  
object1.function1()
```



# ADVANTAGES OF USING CLASSES IN PYTHON

---

- Classes provide an easy way of keeping the data members and methods together in one place which helps in keeping the program more organized.
- Using classes also provides another functionality of this object-oriented programming paradigm, that is, inheritance.
- Classes also help in overriding any standard operator.
- Using classes provides the ability to reuse the code which makes the program more efficient.
- Grouping related functions and keeping them in one place provides a clean structure to the code which increases the readability of the program.



# THE SELF PARAMETER

---

- The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
- It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class.



# Constructors and Destructors

---

Constructors are generally used for instantiating an object. The task of constructors is to initialize (assign values) to the data members of the class when an object of class is created.

In Python the `__init__()` method is called the constructor and is always called when an object is created.


Destructors are called when an object gets destroyed.

The `__del__()` method is known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

# The `__init__()` Function

---

The `__init__` function is a reserved function in classes in Python which is automatically called whenever a new object of the class is instantiated.



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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```





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

    def myfunc(abc):
        print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```



# Inheritance

---

The basic idea of inheritance in object-oriented programming is that a class can inherit the characteristics of another class. The class which inherits another class is called the child class or derived class, and the class which is inherited by another class is called parent or base class.





```
# Create Class Vehicle
class Vehicle:
    def vehicle_method(self):
        print("This is parent Vehicle class method")

# Create Class Car that inherits Vehicle
class Car(Vehicle):
    def car_method(self):
        print("This is child Car class method")
```



```
car_a = Car()
car_a.vehicle_method() # Calling parent class method
```

# Polymorphism

---

The term polymorphism literally means having multiple forms. In the context of object-oriented programming, polymorphism refers to the ability of an object to behave in multiple ways.

Polymorphism in programming is implemented via method-overloading and method overriding.

## Method Overloading

Method overloading refers to the property of a method to behave in different ways depending upon the number or types of the parameters.

# Method Overriding

---

Method overriding refers to having a method with the same name in the child class as in the parent class.

The definition of the method differs in parent and child classes but the name remains the same.



```
# Create Class Vehicle
class Vehicle:
    def print_details(self):
        print("This is parent Vehicle class method")

# Create Class Car that inherits Vehicle
class Car(Vehicle):
    def print_details(self):
        print("This is child Car class method")

# Create Class Cycle that inherits Vehicle
class Cycle(Vehicle):
    def print_details(self):
        print("This is child Cycle class method")
        car_a = Vehicle()
car_a. print_details()

car_b = Car()
car_b.print_details()

car_c = Cycle()
car_c.print_details()
```


# Encapsulation

---

Encapsulation simply refers to data hiding. As a general principle, in object-oriented programming, one class should not have direct access to the data of the other class.

Rather, the access should be controlled via class methods.

To provide controlled access to class data in Python, the access modifiers and properties are used.



```
# Creates class Car
class Car:

    # Creates Car class constructor
    def __init__(self, model):
        # initialize instance variables
        self.model = model

    # Creates model property
    @property
    def model(self):
        return self.__model

    # Create property setter
    @model.setter
    def model(self, model):
        if model < 2000:
            self.__model = 2000
        elif model > 2018:
            self.__model = 2018
        else:
            self.__model = model

    def getCarModel(self):
        return "The car model is " + str(self.model)

carA = Car(2088)
print(carA.getCarModel())

car_a = Car(2088)
print(car_a.get_car_model())
```

# Data Abstraction

---

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonyms because data abstraction is achieved through encapsulation.

```
from abc import ABC
class ClassName(ABC):
```

- An Abstract class can contain the both method normal and abstract method.
- An Abstract cannot be instantiated; we cannot create objects for the abstract class.
- Abstraction is essential to hide the core functionality from the users. We have covered the all the basic concepts of Abstraction in Python.

THANK  
YOU

