Python

→ Created by : Guido Van Rossum

→ released → 1991

→ name inspired by: British Comedy group "Monty Python's flying Circus"

Versions

① Python 1.0 (1991)

↳ functions, Exception handling, data types (list, dic. ---)

② Python 2.0 (2000)

↳ comprehessions & garbage collection

↳ limitations: inconsistency handling in unicode

③ Python 3.0 (2008)

↳ modernized language

2020 → Python 2.0 support officially end

Present day

1. web development
2. Data Analytics
3. Machine learning
4. Automation
5. Gamedevelopment

unicode → support multiple scripts

ASCII ⇒ 'A' ⇒ 65

'a' ⇒ 97

↳ 128

Interpreter vs Compiler

⇒ Both tools used to convert High level programming code (written by Human) to machine code

Interpreter

↳ Reads the program line by line
↳ Easy to find error ⇒ debugging is Easy
↳ Ex: Python, JS, Ruby etc

Cons: Slower

Compiler:

↳ Read the Entire program at once
↳ Convert into machine code (.exe)
↳ Ex: C, C++, Java (partly)

Learn faster

Variables:

⇒ Container that stores data

⇒ It's a Name of memory location which contains the data

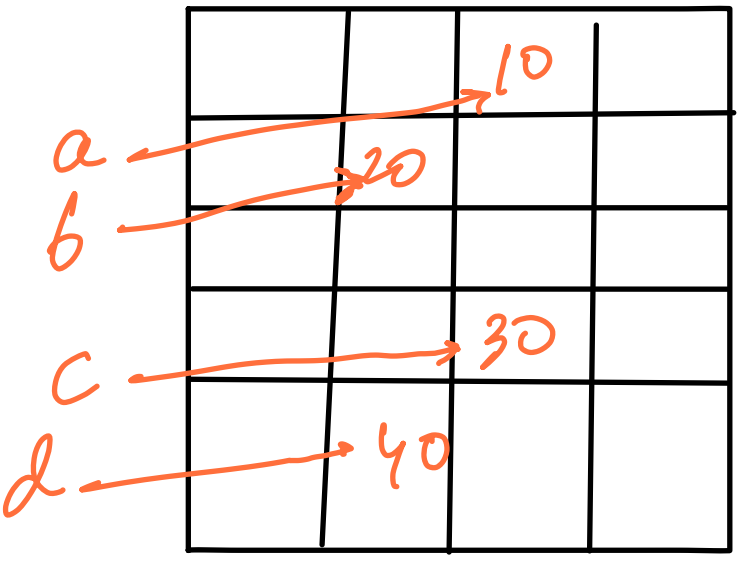
4 bytes

a = 10

b = 20

c = 30

d = 40




```
a = 10  # a is an interger
name = "Arun"   #name is a string
pi = 3.14 # pi is a float
is_active = True # boolean
```

Rules for Naming Variables

① Start with letter or underscore

my_name / _my_name ✓

2my_name (never start with number)

my_name1 ✓

name_1

my-name (No Hyphens)

my name (No space)

② Case Sensitive

Age
age

Both are different

③ Avoid Reserved keywords

Guildline

student_name
student_age

x
y
z (unclear)

studentName (camel casing is not python convention)

Variables are classified based on their scope:

① Local Variable

↳ Define: Inside the function

↳ Scope: within function only

↳ they are destroyed after the function ends

↳ stack


```
def greet(): 1 usage
    message = "Hello Everyone"  # local variable
    print(message)

#print(message)   give error, not accessible outside fuction
greet()
```

2. Global Variable

- Define: outside the function
- Scope: Available throughout the Entire Program
- lifetime: Exist until the program ends
- Heap

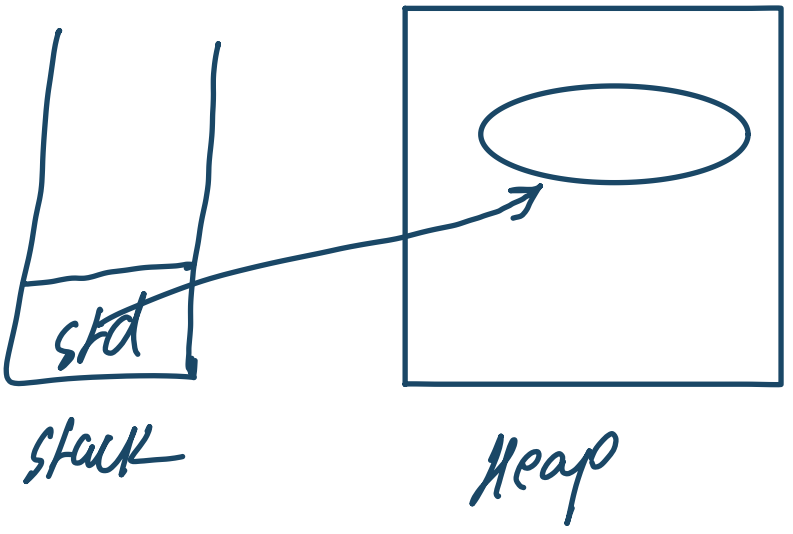
```
name = "Arun"    #Global Variable


def new_greet():  1 usage
    print("Hello", name)
    print(f"Hello, {name}")


new_greet()
```

ref = object

std = Student()

object → heap

ref → stack



DAY 2 (17 Dec)

Modifying global variable

in python create the global variable then create a function and modify that value of global variable

```python
counter = 0  # global variable
def increment():  1 usage
    global counter    # declare to modify global variable
    counter = counter+1
    print("Hello")

increment()
print(counter)  # output 1
```

3. Nonlocal Variable

↳ defined in nested function

```
def outer_function():
    count=0
    def inner_function():
        nonlocal count
        count+=1
```

```python
def outer_function(): 1 usage
    count = 0  #2  # variable in enclosing function

    def inner_function():
        nonlocal count   # refer to the enclosing variable
        count += 1
        print("Count", count)

    inner_function()
    inner_function()

outer_function()
```

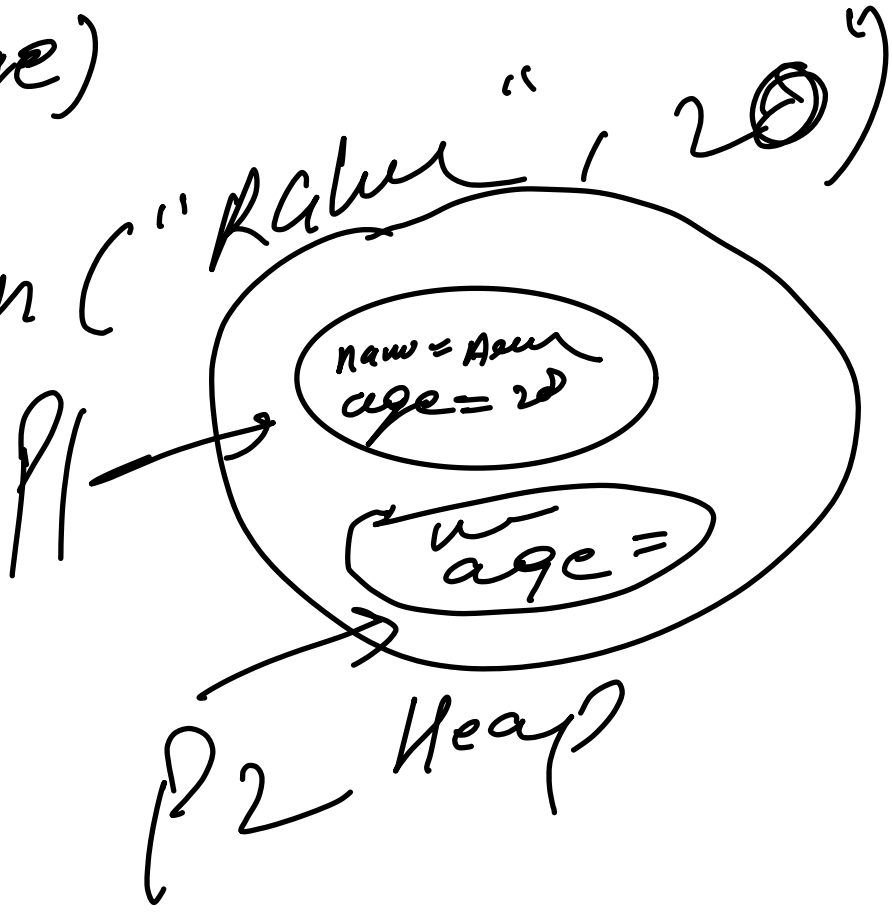
4. Instance Variable

(object specific variable)

↳ tied to specific object

(defined self in classes)

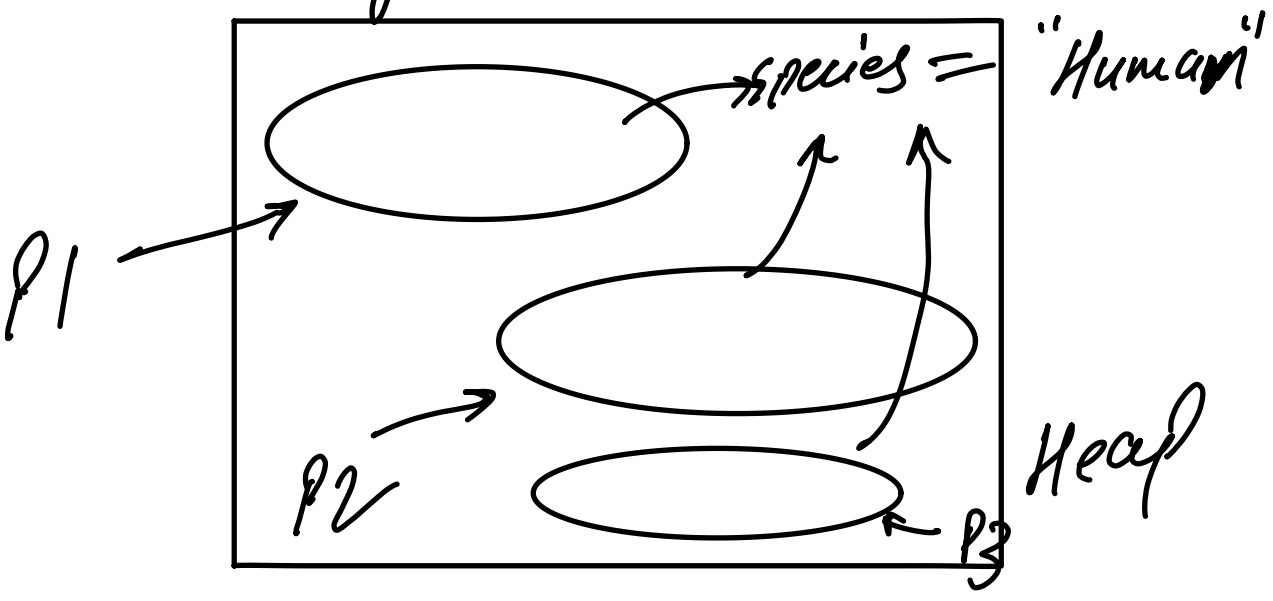
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("Arun", 20)
print(p1.name)
print(p1.age)

p2 = Person("Ravi", 20)
```



(5) Class Variable (Shared variable)

```
class Person:
    species = "Human"   # class Variable

    init function() ---
    ==
    ==

p1 = Person()
p2 = Person()
print(p1.species)
print(p2.species)
```

* Variable shared by all the instances of a class



Type	Defined In	Scope	Lifetime
① local	function	limited to fun	during fun Execution
② Global	outside all fun^n	Entire Prog.	until Program Ends
③ nonlocal	nested fun	Enclosing fun^n	Exists during Enclosing function Execution
④ Instance	inside a class (with self)	specific to an object (PL)	as long as object Exists
⑤ class	inside a class	shared across all objects	as long as class Exists

Datatype

(A) Integer Data types

1. int (Integer)

↳ 5, 10, -20

```
age = 10
print(type(age))
```

<class, 'int'>

② float (Decimal)

pi = 3.14

print(type(pi))

③ Complex ⇒ real + img

z = 3 + 5j

print(type(z))

⑧ Sequence Data types

1. Str (String)
   ↳ ("Hello", 'Arun')
② list : ordered, mutable collection
   ↳ eg: [1, 2, 3, 4]
   ["Arun", "Sumit", "Amit"]
③ tuple : Ordered, immutable collection
   eg ⇒ (1, 2, 3, 4)

C. Set datatype

1. set : unordered, mutable, unique collection

E.g: s = { 1, 2, 3, 4, 3, 2 }

print(s)

② frozenset → Immutable version of set

froze = frozenset({ 1, 2, 3, 4 })

D Mapping Data types

1. dict ⇒ unordered collection of key-value pairs

Key → unique

value → can repeat

Eg: { "name" : "Arun" , "age" : 28 }

E) boolean Datatype

is_active = True
False

```
print("=====integer========")

new_age = 20
print(type(new_age))

print("=====float========")

new_pi = 3.14
print(type(new_pi))

print("=====complex========")

complex = 3+5j
print(type(complex))

print("=====str========")
new_name = "Arun"
print(type(new_name))
```

```python
print("=====list========")
list = ["Arun", "Sumit", "Amit"]
print(type(list))

print("=====tupple========")
tup = ("Arun", "Sumit", "Amit")
print(type(tup))

print("=====set========")
unique_number = {1,2,2,3,4,5,3}
print(unique_number)
print(type(unique_number))

print("=====frozenset========")

froze = frozenset([1,2,2,3,4,6,4])
print(froze)
print(type(froze))
```

```python
print("=====dict=========")
student = {"name":"Arun", "age":28}
print(student)
print(type(student))

print("=====boolean=========")
is_active = True
print(type(is_active))
```