=PYTHON= Mod 1

DATA TYPES → type command to know data type

	Data Type	
Integers	int	Solo enteros positivos o negativos
Numerical/Variable Continuous/Contributive Real numbers	float	Enteros decimales Include integers but also #s in between
Words ① ②	str	
True / False	boolean	

Change the type of the expression "type casting"

Convert

- int to a float → float(2): 2.0
- float to an int → int(1.1): 1
- float to str → str(1): "1"
- 1 to boolean → bool(1): True
- 0 to boolean → bool(0): False
- boolean to int → int(True): 1
- boolean to int → int(False): 0

1 is a Int & 1.1 a float

EXPRESSIONS: Mathematical Operations

Math Symbols

+, -, *, /, // arroja numeros enteros = int

* Son resueltas primero al igual que los () como en algebra

Variables

Assigment operator = or : (etiqueta a mi valor)

my_variable = 10

my_variable: 10

X = 43 + 60 + 16 + 41 Store the result of an expression

x: 160 - Tengo que poner x solita para que arroje el valor

X = 160

y = x/60

bind - atar
tuple - tupla - secuencia

STRINGS " " ' '

Name = "Mike Jack"

String: Slicing

M	I	K	E		J	A	C	K
0	1	2	3	4	5	6	7	8

Name [0:4] = Mike
Name [5:9] = Jack

String: Stride

M	I	K	E	-	J	A	C	K
0	1	2	3	4	5	6	7	8

Name [::2] : "MK-AK"
Indicates every second variable
Name [0:5:2] : "MK_"
Indicates every second value upto 5

String: lenght
Len("Mike Jack") 8

String -> Concatenate

Name = "MK"
Name + "bueno"
"MK bueno"

Escape Sequences
\n Resultado en 2 lineas
\t Resultado con espacio mismo renglon tab
r" or \\ Solo agrega 1 raya a mi texto Ale\Rami

STRING METHODS

Replace
B = Name.replace('Mike','Ale')
B: "Ale Jack"

Find
Name.find('Mi') : 0
Name.find('Ke') : 2

= PYTHON = Mod 2.

* Porque se vale se usan las listas si es más fácil trabajar con ellas

List & Tuples (secuencias)

Tuple 1 = ("disco", 10, 1.2)
0 1 2
-3 -2 -1

Tuple1[0] : "disco"
Tuple1[-3] : "disco"

* Aplica igual en una lista

Slicing
Tuple1[0:2] : ("disco", 10) * Aplica igual a una lista

Lenght
Len(("disco", 10, 1.2) = 3 * Aplica igual a una lista

List mutable -> the original can be changed
Tuples immutable -> the original cannot be changed

List
L = ["Mike Jack", 10.1, 1982]
L.extend ["pop", 10] * Original list is modified
["Mike Jack", 10.1, 1982, "pop", 10] Len = 5
L.append ["pop", 10]
["Mike Jack", 10.1, 1982, ["pop, 10]] Len = 4
L[0] = "Ale Rami"
L = ["Ale Rami", 10.1, 1982]
del (L[0]) -> delete "Ale Rami"
del (L[2]) -> delete 1982
"Ale Rami".split() : ['Ale', 'Rami']
"A,B,C,D".split(,) : ['A', 'B', 'C', 'D']

Nesting
NT = (1, 2, ("pop", "rock"), (3,4), ("disco", (1,2)))
0 1 2 3 4
NT[2] : ("pop", "rock")
NT[2][0]
NT[2][1]
Acceso 1 elemento de 2
Acceso 2 elemento de 2

SETS

- Sets are a type of collection
  Like list and tuples you can input different Python types
- Unlike list and tuples they are unordered
  Do not record element positions
- Set only have unique elements
  Only one of a particular element in a set

curly brackets for sets

```
Set1 = {" pop", "rock", "soul", "hard rock"}
```

Convert a list to a set -> Command `Set()`

```
alb_list = ["Mike Jack", "Thriller", "Thriller", 1982]
alb_set = set(alb_list)        ← Solo 1
alb_set : {'Mike Jack', 'Thriller', 1982}
```

Set Operations

```
A = {'a', 'b', 'c'}
A.add('z')
A: {'a', 'b', 'c', 'z'}

A = {'a', 'b', 'c'}
A.remove('b')
A: {'a', 'c'}
```

```
A: {'a', 'b', 'c'}
'c' in A
True
'z' in A
False
```

```
alb_set1.difference(alb_set2)
{'a', 'c'}
alb_set2.difference(alb_set1)
{'d', 'f'}
```

Set Operations - Mathematical

1. ```
   alb_set1 = {'a', 'b', 'c'}
   alb_set2 = {'d', 'f', 'b'}
   alb_set3 = alb_set1 & alb_set2
   alb_Set3 : {'b'}
   ```

2. ```
   alb_Set1.intersection(alb_set2)
   {'b'}
   ```

```
alb_set1.union(alb_set2)
{'a', 'b', 'c', 'd', 'f'}

alb_set3.issubset(alb_set1)
True        subset
```

DICTIONARIES

- Dictionaries are denoted with curly brackets {}
- The keys have to be immutable and unique
- The values can be immutable, mutable and duplicates
- Each key and value pair is separated by a comma

```
DICT {'Key1': 1, 'Key2': '2', 'Key3': [3,3,3], 'Key4': (4,4,4), 'Key5': 5}
      Key   Value
```

Value pair

```
DICT['Key1']: 1
del(DICT['Key1'])   -> It'll remove 'Key1' and value 1 from the dictionary
'Key2' in DICT      'Key20' in DICT
True                False
DICT.keys() = ['Key1', 'Key2', 'Key3', 'Key4', 'Key5']   - da lista de keys
DICT.values() = [1, "2", [3,3,3], (4,4,4), 5]
```

add an entry

```
DICT['Key10'] = '10'
DICT
```

= PYTHON = Mod 3

CONDITIONS AND BRANCHING

Comparison Operators

a = 6 → Asigno el valor de 6 a a

a == 7 → doble signo igual es para comparar igual ¿6 = 7? Equality

False

a = 6 * También se puede hacer con strings

a == 6 a = Ale

True a == Ale True

i = 6 → Asigno el valor de 6 a a Inequality

i != 6 → ! = no es igual a

False → 6 es igual a 6, es falso porque no cumplen la desigualdad

IF statement

```
age = 19
if (age > 18):
    print ("you can enter")
print ("move on")
```

```
age = 17
if (age > 18):
    print ("you can enter")
print ("move on")
```

The statements after the if statement will run regardless if the condition is true or false.

ELSE statement

```
age = 19
if (age > 18):
    print ("you can enter")
else:
    print ("go see Meat Loaf")
print ("move on")
```

```
age = 17
if (age > 18):          # if true
    print ("you can enter")
else:                   # if false
    print ("go see Meat Loaf")
print ("move on")
```

ELIF statement (else if)

Allow us to check additional conditions

age = 18 age = 17 age = 19

```
if (age > 18)
    print ("you can enter")
elseif (age == 18)
    print ("go see Pink Floyd")
else: → Se ejecuta si nada de arriba es true
    print ("go see Meat Loaf")
print ("move on")
```

LOGIC OPERATORS

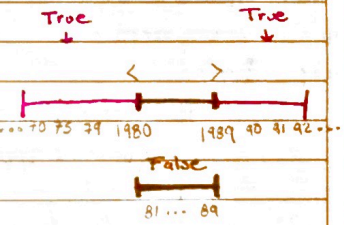
not (True) not (False)

False True

or

alb_yr = 1990 alb_yr = 1981

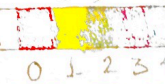
```
if (alb_yr < 1980) or (alb_yr > 1989)
    print ("Album made in the 70's or 90's")
else:
    print ("Album made in the 1980's")
```



LOOPS

- Range function — Index: range (3): [0,1,2] range (10,15): [10,11,12,13,14] (+5)
- For loops

Can be used on tuples and lists. Loops perform a task over and over



5 iterations

Squares = ["red", "yellow", "green", "purple", "blue"]

```
for i in range (0,5):
    squares[i] = "White"
```

i → Is set to zero and it'll iterate 5 times

Indent / Sangría

Norma

We can also iterate through a list or tuple directly in Python, we do not even need to use indexes.

0 1 2

```
Squares = ['red', 'yellow', 'green']
for square in squares:        → Nombre de la lista
    square
```

0 = first iteration = red
1 = 2nd iteration = yellow
2 = 3rd iteration = green

```
Squares = ['red', 'yellow', 'green']
for i, square in enumerate(squares):
    square
    i
```

Square	i
'red'	0
'yellow'	1
'green'	2

While Loops → Only runs if a condition is met

```
Interestante square = ['orange', 'orange', 'purple', 'blue']
nueva lista Newsquares = []
index  i = 0
while (squares[i] == 'orange'):
    Newsquares.append(squares[i])
    i = i + 1
```

Squares
i 0 1 2 3 4

while (squares[i] == 'orange'):

New squares

Si es naranja hace la sig iteracion y le suma un 1 a i, se detiene solo si la condición no se cumple

FUNCTIONS

ej. Len(), Set(), sum(), etc... { Built-in funtions

Functions	vs	Methods
Functions take an input and produce a new output		Methods don't produce a new output, they modify the input
alb_rating = [1,3,2,6,5,4]		alb_rating = [1,3,2,6,5,4]
sorted_alb_rating = sorted(alb_rating)		alb_rating.sort()
sorted_alb_rating = [1,2,3,4,5,6]		alb_rating = [1,2,3,4,5,6]

Norma

(alb_rating) → 2 listas c/nombre diferente

[alb_rating] → 1 lista única

MAKE FUNCTIONS

Nombre de la función
dos! sabemos que es una función → Parámetro

```
def add1(a):
    """add 1 to a"""        ← Documentation string = help command
    b = a + 1               ← Code block (body)
    return b
```

sangría

Python doesn't allow a function to have an empty body

MULTIPLE PARAMETERS

2 parámetros

```
def Mult(a, b):
    c = a * b
    return c
```

float * int = 10 * 2.1 = 21
str * int = 2 * "Mike" = 'MikeMike'
int * str = 2 * '1' = '1' '1'
↳ We won't get an error

If the return statement is not called, Python will automatically return a NONE re.

```
def NoWork():
    pass
    return None
```

SCOPE

Global Scope - Variables can be defined in the global scope, meaning that are accesible everywhere

Local Scope - Local variables only exist within the scope of a function
Variable inside the global scope can have the same name as variables in the local scope w/o conflict
* If a variable is not defined within a function, Python will check the global scope.

```
def Thriller():
    Date = 1982          → Variable with local scope
    return (Date)
Date = 2017              → Varible with global scope
print (Thriller())   Resp: = 1982   → We're printing the date of the function
print (Date)         = 2017         → We're printing just the global variable
```

Norma

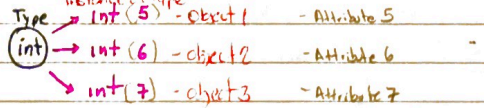
```
def Pink-Floyd():
    global Claimed Sales          We can tell python if a variable is global
    Claimed Sales = '45 millons'  just by adding the keyword global
    return ClaimedSales

PinkFloyd()
print (ClaimedSales)
'45 millons'
```

OBJECTS AND CLASSES

Class or type is the same

Data Types: Int: 1,2,3 ; float: 1.1, 2.1, 3.1 ; String: 'abc', 'Hola';
(they're also objects) List: [1, 2, 'abc']; Dictionary: {'dog': 1, 'cat': 2} ;
Bool: False, True

ie. Everytime we create an integer we're creating an instance of type integer or we're creating an integer object.



type() → Command to know the type of an object

```
>> type([1,34,3])      >> type(1)          >> type('Hello')
<Class 'list'>         <Class 'int'>       <Class 'str'>
```

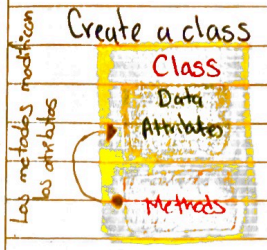
Methods

- Sorting is an example of a method that interacts with the data in the object

```
Ratings = [1, 4, 3, 2]
Ratings.sort()
```

Class

In Python we can create our own class or type



Create a class

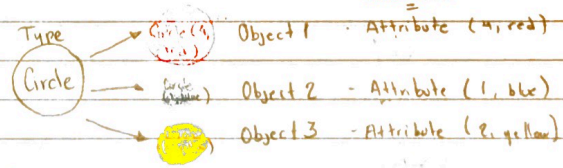
Object 1 — The class has data attributes
Object 2 — The class has methods
Object 3 — We create instances of that class or objects
Object 4

```
class Circle (object):
```

(2) (1) (3)

Object 1: Instance of type Circle
Data Attributes
radius - 4
color - red

1 Class Definition: This tells python, I'm creating my own class
2 Name of class: El nombre que yo le asigno
3 Class parent: Siempre se deja la palabra object




```
class Circle (object):          Parameters        Define your class
    def __init__(self, radius, color):
        self.radius = radius;
        self.color = color;
```

The parameters are used to initialize the radius and color data attributes of the class instance

Data attributes used to initialize each instance of the class.

(1) → Self es como una caja que guarda los data attributes
__init__ → special method or constructor used to initialize data attributes

```
    def add_radius(self, r):
        self.radius = self.radius + r
```

Method used to add r to radius

```
C1 = Circle(2, 'red')
C1.add_radius(8)
return (10, 'red')
```

C1 → Mi variable se llama C1, mis parametros los defino como 2 radio y 'red' color.
C1.add_radius(8) → Estoy llamando a mi metodo para que sume 8 al radio