¿Qué es Python?

- Es un lenguaje de programación de muy alto nivel. Debido a sus tipos de datos más generales es aplicable a más dominios.
- Es un lenguaje interpretado, lo cual permite ahorrarte mucho tiempo durante el desarrollo.

¿Porqué usar Python?

Los programas en Python son tipicamente más cortos que sus programas equivalentes en otros lenguajes por varios motivos:
- Los tipos de datos de alto nivel permiten expresar operaciones complejas en una sola instrucción.
- La agrupación de instrucciones se hace mediante indentación en vez de llaves de apertura y cierre.
- No es necesario declarar variables ni argumentos.

***** El lenguaje recibe su nombre del programa de televisión de la BBC "Monty Python's Flying Circus"***

Interprete Python

- Path:

```
/usr/local/bin
```

- Command Line:

python -c comando [arg] …
python -c "a='example';print a"

python -m module [arg] …
Ejecuta un módulo como un programa.
python -m timeit "'-'.join(str(n) for n in range(100))'

python -i script.py
Ejecutar script y luego ingresar al modo interactivo.

- Paso de argumentos:

```
import sys
sys.argv[0]
```

- Encoding:

```
#!/usr/bin/env python3                    shebang
# -*- coding: encoding -*-
```

Introducción Informal

1) Números

- Operadores: +, -, *, /

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

- División

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

- Potencia

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

En el modo interactivo, la última expresión impresa se asigna a la variable _.

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

2) Cadena Caracteres

```
>>> "'Isn\'t," they said.'
"'Isn\'t," they said.'
>>> print("'Isn\'t," they said.')
"Isn't," they said.
>>> s = 'First line.\nSecond line.'
# \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'ununununium'
```

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

```
>>> prefix + 'thon'
'Python'
```

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
```

```
'P'


>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Una forma de recordar cómo funcionan los *slicing* es pensar que los índices apuntan *entre* caracteres.

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

3) Listas

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]


>>> squares[:]
[1, 4, 9, 16, 25]
```

A diferencia de las cadenas, las listas son mutables.

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

4) Primeros pasos hacia la programación:

```
>>> a, b = 0, 1
>>> while a < 1000:
...     print(a, end=',')
...     a, b = b, a+b
...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Más herramientas para el control de flujo.

1) Sentencia IF

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

2) Sentencia FOR

```
# Strategy:  Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy:  Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

3) Función range()

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4


range(5, 10)
   5, 6, 7, 8, 9
range(0, 10, 3)
   0, 3, 6, 9
range(-10, -100, -30)
  -10, -40, -70


>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

4) Break, continue, else

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

La cláusula else se ejecuta cuando no se genera ninguna excepción, y el else de un bucle se ejecuta cuando no hay ningún break.

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found an odd number", num)
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

5) pass

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

6) Funciones

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

De hecho, técnicamente hablando, los procedimientos sin return sí retornan un valor, aunque uno aburrido. Este valor se llama **None** (es un nombre predefinido).

6.1) Argumentos con valores por omisión

Los valores por omisión son evaluados en el momento de la definición de la función, en el ámbito de la definición, entonces:

```
i = 5
def f(arg=i):
    print(arg)

i = 6
f()
5
```

Existe una diferencia cuando el valor por omisión es un objeto mutable como una lista, diccionario, o instancia de la mayoría de las clases.

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
[1]
print(f(2))
[1, 2]
print(f(3))
[1, 2, 3]
```

Si no se quiere que el valor por omisión sea compartido entre subsiguientes llamadas, se pueden escribir la función así:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

6.2) Palabras claves como argumentos

```
def parrot(voltage, state='a stiff', action='voom',
type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

a) Forma Correcta

```
# 1 positional argument
parrot(1000)
# 1 keyword argument
parrot(voltage=1000)
# 2 keyword arguments
parrot(voltage=1000000, action='VOOOOOM')
# 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)
# 3 positional arguments
parrot('a million', 'bereft of life', 'jump')
# 1 positional, 1 keyword
parrot('a thousand', state='pushing up the daisies')
```

b) Forma Incorrecta

```
# required argument missing
parrot()
# non-keyword argument after a keyword argument
parrot(voltage=5.0, 'dead')
# duplicate value for the same argument
parrot(110, voltage=220)
# unknown keyword argument
parrot(actor='John Cleese')
```

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])

cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

6.3) Parámetros especiales

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |           |                |
        |       Positional or keyword   |
        |                               - Keyword only
         -- Positional only
```

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *,
kwd_only):
...     print(pos_only, standard, kwd_only)
```

***Finalmente, considere esta definición de función que contiene una colisión potencial entre los parámetros posicionales name y **kwds que incluye name como una clave:*

```
def foo(name, **kwds):
    return 'name' in kwds
```

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

```
def foo(name, /, **kwds):
    return 'name' in kwds
>>> foo(1, **{'name': 2})
True
```

Para una API, utilice únicamente posicionales para prevenir cambios que rompan con la compatibilidad de la API si el nombre del parámetro es modificado en el futuro.

6.4) Lista de argumentos arbitrarios

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

6.5) Desempaquetando una lista de argumentos

```
>>> list(range(3, 6))            # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))            # call with arguments unpacked from a list
[3, 4, 5]
```

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
```

6.6) Expresiones lambda

Pequeñas funciones anónimas pueden ser creadas con la palabra reservada lambda.

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

6.7) Documentación

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.

    No, really, it doesn't do anything.
```

6.8) Anotaciones

Las anotaciones de funciones son información completamente opcional sobre los tipos usadas en funciones definidas por el usuario

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

7) Coding Style

Para Python, PEP 8 se erigió como la guía de estilo a la que más proyectos adhirieron; promueve un estilo de codificación fácil de leer y visualmente agradable. Todos los desarrolladores Python deben leerlo en algún momento

- Usar sangrías de 4 espacios, no tabuladores.
- Recortar las líneas para que no superen los 79 caracteres.
- Usar líneas en blanco para separar funciones y clases, y bloques grandes de código dentro de funciones.
- Cuando sea posible, poner comentarios en una sola línea.
- Usar docstrings.
- Usar espacios alrededor de operadores y luego de las comas, pero no directamente dentro de paréntesis: a = f(1, 2) + g(3, 4).
- Nombrar las clases y funciones consistentemente; la convención es usar NotacionCamello para clases y minusculas_con_guiones_bajos para funciones y métodos.
- Siempre usa self como el nombre para el primer argumento en los métodos.
- El default de Python, UTF-8, o incluso ASCII plano funcionan bien en la mayoría de los casos.

Estructuras de datos

1) Listas

- list.append(x)

Equivale: a[len(a):] = [x]

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.append(‘grape’)
['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana', 'grape']
```

- list.extend(iterable)

```
>>> fruits.extend([‘strawberry’])
['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana', 'grape', 'strawberry']
```

- list.insert(i, x)
Equivale: a.insert(len(a), x) == a.append(x)
>>> fruits.insert(0, ‘pineapple’)
['pineapple', 'orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana', 'grape', 'strawberry']

- list.remove(x)
Quitar el primer item de la lista cuyo valor sea x.
>>> fruits.remove('kiwi')
['pineapple', 'orange', 'apple', 'pear', 'banana', 'apple', 'banana', 'grape', 'strawberry']

- list.pop(i)
>>> fruits.pop(-3)
'banana'
El parámetro i es opcional.
Si no se especifica un índice quita y retorna el último item de la lista.
>>> fruits.pop()
'strawberry'
['pineapple', 'orange', 'apple', 'pear', 'banana', 'apple', 'grape']

- list.clear()
Equivale: del a[:]

- list.index(x, start, end)
Retorna el índice del primer elemento cuyo valor sea igual a x.
Los parámetros start y end son opcionales.
>>> fruits.index('apple')

- list.count(x)
>>> fruits.count('apple')
2

- list.sort(key=None, reverse=False)
>>> fruits.sort()
['apple', 'apple', 'banana', 'grape', 'orange', 'pear', 'pineapple']

- list.reverse()
>>> fruits.reverse()
['pineapple', 'pear', 'orange', 'grape', 'banana', 'apple', 'apple']

- list.copy()
Equivale: b = a[:]

1.1) Usando lista como pilas.
Último en entrar, primero en salir
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]

1.2) Usando listas como colas.
Primero en entrar, primero en salir.
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")
>>> queue.append("Graham")
>>> queue.popleft()
'Eric'
>>> queue.popleft()
'John'
>>> queue
deque(['Michael', 'Terry', 'Graham'])

1.3) Compresión de listas
Manera concisa de crear listas.
Ejemplo: crear una lista de cuadrados:
Forma1:
>>> squares = []
>>> for x in range(10):
... squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Forma2:
>>> squares = list(map(lambda x: x**2, range(10)))

Forma3:
>>> squares = [x**2 for x in range(10)]

Más ejemplos:
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
             ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]

1.4) Listas por comprensión anidadas.
>>> matrix = [
... [1, 2, 3, 4],
... [5, 6, 7, 8],
... [9, 10, 11, 12],
...]

>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

**En el mundo real, deberías preferir funciones predefinidas a declaraciones con flujo complejo.
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]

2) Instrucción del

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` puede usarse también para eliminar variables:

```
>>> del a
```

3) Tuplas y secuencias

Las tuplas son **inmutables**.

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Las tuplas vacías se construyen con un par de paréntesis vacíos.
Una tupla con un item se construye poniendo una coma a continuación del valor.

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

4) Conjuntos

Es una colección no ordenada y sin elementos repetidos.

Usos básicos: verificación de pertenencia y eliminación de entradas duplicadas.
Soportan operaciones matemáticas como la unión, intersección, diferencia y diferencia simétrica.

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Comprensión de conjuntos:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5) Diccionarios

Los diccionarios se indexan con claves, que puede ser cualquier tipo inmutable.
Las cadenas y números siempre pueden ser claves.
Las tuplas pueden usarse como claves si solamente contienen cadenas, números o tuplas.

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False

>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Comprensiónes de diccionarios:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Cuando las claves son cadenas simples:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

6) Técnicas de iteración

- items()

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
```

```
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

- enumerate()

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

- zip()

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

- reversed()

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

- sorted()

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

- set()

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

- Es mejor crear una nueva lista a modificar una mientras se está iterando:

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

7) Más acerca de condiciones

```
>>> a = 10
>>> b = 12
>>> c = 12
>>> bool(a < b == c)
True
```

short-circuit operators: and, or

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
>>> string2 and string3
'Hammer Dance'
```

Walrus operator :=

```
data = [1,2,3,4]
f_data = [y for x in data if (y := f(x)) is not 4]
```

*https://towardsdatascience.com/when-and-why-to-use-over-in-python-b91168875453

8) Comparando secuencias y otros tipos

La comparación usa orden lexicográfico.

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)              == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Módulos:

file: fibo.py

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Importar módulo desde el interprete:

```
>>> import fibo
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Un módulo puede contener tanto **declaraciones ejecutables** como **definiciones de funciones**.
Las declaraciones están pensadas para inicializar el módulo. Se ejecutan únicamente la primera vez que el

módulo se encuentra en una declaración import. También se ejecutan si el archivo se ejecuta como script.

Esto no introduce en el espacio de nombres local el nombre del módulo desde el cual se está importando.
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

- *
Esto importa todos los nombres excepto los que inician con un guión bajo (_).
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
La mayoría de las veces los programadores de Python no usan esto ya que introduce en el intérprete un conjunto de nombres desconocido.

- as
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

1) Ejectuando módulos como scripts:
python fibo.py <arguments>

El código en el módulo será ejecutado, tal como si lo hubieses importado, pero con __**name**__ con el valor de "__**main**__"

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

El código que analisa la línea de órdens sólo se ejecuta si el módulo es ejecutado como archivo principal.
$ python fibo.py 50

0 1 1 2 3 5 8 13 21 34

Si el módulo se importa, ese código no se ejecuta:
>>> import fibo
>>>

2) Ruta de búsqueda de módulos
a) El directorio que contiene el script de entrada (o el directorio actual cuando no se especifica archivo).
b) PYTHONPATH (una lista de nombres de directorios, sys.path).
c) La instalación de dependencias por defecto.

3) Compilado de Python
Para acelerar la carga de módulos, Python cachea las versiones compiladas de cada módulo en el directorio __**pycache**__ bajo el nombre **module.version.pyc**

Python chequea la fecha de modificación de la fuente contra la versión compilada. Este es un proceso completamente automático.

Python no chequea la caché si el módulo es cargado directamente desde la línea de comandos.

4) Módulos estándar
Algunos módulos se integran en el intérprete; estos proveen acceso a operaciones que no son parte del núcleo del lenguaje pero que sin embargo están integrados, tanto por eficiencia como para proveer acceso a primitivas del sistema operativo, como llamadas al sistema.
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>

5) Función dir()
La función integrada dir() se usa para encontrar qué nombres define un módulo.

>>> import fibo, sys

>>> dir(fibo)
['__name__', 'fib', 'fib2']

6) Paquetes
Los Paquetes son una forma de estructurar el espacio de nombres de módulos de Python usando «nombres de módulo con puntos». Por ejemplo, el nombre del módulo A.B designa un submódulo B en un paquete llamado A.

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
            __init__.py
            wavread.py
            wavwrite.py
            aiffread.py
            aiffwrite.py
            auread.py
            auwrite.py
            ...
      effects/                  Subpackage for sound effects
            __init__.py
            echo.py
            surround.py
            reverse.py
            ...
      filters/                  Subpackage for filters
            __init__.py
            equalizer.py
            vocoder.py
            karaoke.py
            ...
```

Los archivos __init__.py son obligatorios para que Python trate los directorios que contienen los archivos como paquetes.

Importando módulos:
a)
import sound.effects.echo

sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)

b)
from sound.effects import echo
echo.echofilter(input, output, delay=0.7, atten=4)

c)
from sound.effects.echo import echofilter
echofilter(input, output, delay=0.7, atten=4)

La sintaxis como `import` **`item.subitem.subsubitem`**, cada ítem excepto el último debe ser un paquete; el mismo puede ser un módulo o un paquete pero **no puede ser una clase, función o variable** definida en el ítem previo.

6.1) Importando * desde un paquete.
La declaración `import` usa la siguiente convención: si el código del **`__init__.py`** de un paquete define una lista llamada **`__all__`**, se toma como la lista de los nombres de módulos que deberían ser importados cuando se hace **`from package import *`**.

Por ejemplo, el archivo **`sound/effects/__init__.py`** podría contener el siguiente código:
__all__ = ["echo", "surround", "reverse"]
Esto significaría que **`from sound.effects import *`** importaría esos tres submódulos del paquete `sound`.

Si no se define **`__all__`**, la declaración **`from sound.effects import *`** ***no*** **importa todos los submódulos del paquete `sound.effects` al espacio de nombres actual**; sólo se asegura que se haya importado el paquete `sound.effects`

¡no hay nada malo al usar `from package import specific_submodule`! De hecho, esta es la notación recomendada.

6.2) Referencias internas en paquetes
Import absoluto:
Si el módulo **`sound.filters.vocoder`** necesita usar el módulo **`echo`** en el paquete **`sound.effects`**, puede hacer **`from sound.effects import echo`**.

Import relativos:
from . import echo
from .. import formats
from ..filters import equalizer

Entrada y salida

Maneras de escribir valores:
Declaraciones de expresiones y la función print()
Una tercera manera es usando el método write() de los objetos tipo archivo.

1) Formateo elegante de la salida
Más control sobre el formato de salida:
- Formatted string literals

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- str.format()

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

- Operaciones de concatenación , segmentación de cadena y operaciones propias de las cadenas.

* Visualización rápida de algunas variables con fines de depuración:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

1.1) Formatear cadenas literales

Formatted string literals, también llamados **f-strings.**

- Expresión seguida de un especificador de formato.

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

- Pasar un entero después de **':'**

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

- '!a' aplica ascci(), '!s' aplica str(), '!r' aplica repr()

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
```

My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.

1.2) Método format()

>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"

- Referencia por nombre de argumento.
>>> print('This {food} is {adjective}.'.format(
... food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.

- Combinando argumentos posicionales y nombrados.
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred', other='Georg'))
The story of Bill, Manfred, and Georg.

- Notación **
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678

- Pasar un entero después de **':'**
>>> for x in range(1, 11):
... print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

1.3) Formateo manual de cadenas

-El ejemplo anterior formaeado manualmente:
>>> for x in range(1, 11):
... print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
... # Note use of 'end' on previous line
... print(repr(x*x*x).rjust(4))
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

- str.zfill()
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'

1.4) Viejo formateo de cadenas

- 'string' % values
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.

2) Leyendo y escribiendo archivos

La función **open()** retorna un **file object.**

Se usa pasando dos argumentos:
open(nombre_de_archivo, modo)

```
>>> f = open('workfile', 'w')
```

mode:
- ‘r’ → solo lectura.
- ’w’ → solo escritura.
- ‘a’ → abre el fichero para agregar.
- ‘r+’ → lectura y escritura.
- si se omite, por defecto es ‘r’

Normalmente los ficheros se abren en modo texto.
‘b’ agregado al modo abre el fichero en modo binario, y los datos se leerán y escribirán en forma de objetos de bytes.
Este modo debería usarse en todos los ficheros que no contienen texto.

Cuando se escribe en modo texto, por defecto se convierten los \n a los finales de linea específicos de la plataforma.
Este cambio automático está bien para archivos de texto, pero corrompería datos binarios como los de archivos JPEG o EXE. Asegurate de usar modo binario cuando leas y escribas tales archivos.

>>> with open('workfile') as f:
... read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True

2.1) Métodos de los objetos archivo

- Leer un archivo.
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''

- Leer una sola línea del archivo.

>>> f.readline()

'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''

- Otra forma de leer líneas de un archivo más eficiente.
>>> for line in f:
... print(line, end='')
...
This is the first line of the file.
Second line of the file

* Si querés leer todas las líneas de un archivo en una lista también podés usar list(f) o f.readlines().

```
>>> f.write('This is a test\n')
15
```

- f.tell() retorna un entero que indica la posición actual en el archivo representada como número de bytes desde el comienzo del archivo en modo binario y un número opaco en modo texto.

- Para cambiar la posición del objeto archivo, utiliza f.seek(offset, whence). La posición es calculada agregando el *offset* a un punto de referencia; el punto de referencia se selecciona del argumento *whence*.

- Un valor *whence* de 0 mide desde el comienzo del archivo, 1 usa la posición actual del archivo, y 2 usa el fin del archivo como punto de referencia.

- *whence* puede omitirse, el valor por defecto es 0, usando el comienzo del archivo como punto de referencia.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

`2.2) Guardando datos estructurados con json`

Sin embargo, cuando querés grabar tipos de datos más complejos como listas, diccionarios, o instancias de clases, las cosas se ponen más complicadas.

JSON (JavaScript Object Notation)

El módulo estándar llamado json puede tomar datos de Python con una jerarquía, y convertirlo a representaciones de cadena de caracteres; este proceso es llamado *serializing*.

Reconstruir los datos desde la representación de cadena de caracteres es llamado *deserializing*.

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Otra variante de la función dumps(), llamada dump(), simplemente serializa el objeto a un archivo de texto. Así que, si f es un objeto archivo de texto abierto para escritura, podemos hacer:

```
json.dump(x, f)
```

Para decodificar un objeto nuevamente, si f es un objeto archivo de texto que fue abierto para lectura:

```
x = json.load(f)
```

Errores y excepciones

1) Errores de sintaxis

Son errores de interpretación

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
               ^
SyntaxError: invalid syntax
```

2) Excepciones

Incluso si una declaración o expresión es sintácticamente correcta, puede generar un error cuando se intenta ejecutar.

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

3) Gestionando excepciones

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try
again...")
...
```

a) Primero, se ejecuta la cláusula *try* (la(s) linea(s) entre las palabras reservadas try y la except).

b) Si no ocurre ninguna excepción, la cláusula *except* se omite y la ejecución de la cláusula try finaliza.

c) Si ocurre una excepción durante la ejecución de la cláusula *try* el resto de la cláusula se omite. Entonces, si el tipo de excepción coincide con la excepción indicada después de la except, la cláusula *except* se ejecuta, y la ejecución continua después de la try.

d) Si ocurre una excepción que no coincide con la indicada en la cláusula *except* se pasa a los try más externos; si no se encuentra un gestor, se genera una *unhandled exception* (excepción no gestionada) y la ejecución se interrumpe con un mensaje como el que se muestra arriba.

** Un *except* puede nombrar múltiples excepciones usando paréntesis, por ejemplo:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

- Una clase en una cláusula `except` es compatible con una excepción si es de la misma clase o de una clase derivada de la misma.

- Pero de la otra manera, una clase derivada no es compatible con una clase base.

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Nótese que si las cláusulas *except* estuvieran invertidas (con `except B` primero), habría impreso B, B, B — se usa la primera cláusula *except* coincidente.

```
B != C; B != D
C == B; C != D
D == C; D == B
```

El último *except* puede omitir el nombre de la excepción capturada y servir como comodín.

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

La cláusula *except* puede especificar una variable después del nombre de excepción. La variable se vincula a una instancia de la excepción con los argumentos almacenados en `instance.args`.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed
directly,
...                          # but may be overridden in exception
subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Los gestores de excepciones no se encargan solamente de las excepciones que ocurren en el *bloque try*.

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

4) Lanzando excepciones

La declaración `raise` permite al programador forzar a que ocurra una excepción específica.

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

Una versión simplificada de la instrucción `raise` sin intención de gestionarla.

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

5) Encadenamiento de excepciones (Exception chaining)

```
>>> def func():
...     raise IOError
...
>>> try:
...     func()
... except IOError as exc:
...     raise RuntimeError('Failed to open database') from
exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
OSError

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
```

** El encadenamiento de excepciones sucede dentro del bloque **except** o **finally.**

** Se puede deshabilitar usando **from None**

```
try:
    open('database.sqlite')
except IOError:
    raise RuntimeError from None
```

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

6) Excepciones definidas por el usuario

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error
occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition
that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition
is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

7) Definiendo Clean-up Actions

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

** La cláusula finally se ejecuta independientemente de que la cláusula try produzca o no una excepción.

a) Si la excepción no es gestionada por una cláusula except, la excepción es relanzada después de que se ejecute el bloque de la cláusula finally.

b) Si el bloque try llega a una sentencia **break**, **continue** o **return**, la cláusula finally se ejecutará justo antes de la ejecución de dicha sentencia.

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False

>>> def divide(x, y):


...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

8) Predefinidas Clean-up Actions

Algunos objetos definen acciones de limpieza estándar para llevar a cabo cuando el objeto ya no necesario, independientemente de que las operaciones sobre el objeto hayan sido exitosas o no.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

Clases

Las clases proveen una forma de empaquetar datos y funcionalidad juntos.

Al crear una nueva clase, se crea un nuevo *tipo* de objeto, permitiendo crear nuevas *instancias* de ese tipo.

1) Ámbitos y espacios de nombres en Python

Un *espacio de nombres* es una relación de nombres a objetos.

En la expresión modulo.funcion, modulo es un objeto módulo y funcion es un atributo de éste.

Un *ámbito* es una región textual de un programa en Python donde un espacio de nombres es accesible directamente.

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"
```

```
    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

El resultado del código ejemplo es:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Notá como la asignación *local* (que es el comportamiento normal) no cambió la vinculación de *algo* de *prueba_ambitos*. La asignación **nonlocal** cambió la vinculación de *algo* de *prueba_ambitos*, y la asignación **global** cambió la vinculación a nivel de módulo.

También podés ver que no había vinculación para *algo* antes de la asignación **global**.

Los objetos módulo tienen un atributo de sólo lectura secreto llamado `__dict__` que retorna el diccionario usado para implementar el espacio de nombres del módulo.

El nombre `__dict__` es un atributo pero no un nombre global.

2) Un primer vistazo a las clases

2.1) Sintaxis de definición de clases

```
class ClassName:

    <statement-1>
    .
    .
    .
    <statement-N>
```

Cuando se ingresa una definición de clase, se crea un nuevo espacio de nombres, el cual se usa como ámbito local; por lo tanto, todas las asignaciones a variables locales van a este nuevo espacio de nombres.

2.2) Objetos clase

Los objetos clase soportan dos tipos de operaciones: hacer referencia a atributos e instanciación.

Para *hacer referencia a atributos* se usa la sintaxis estándar de todas las referencias a atributos en Python: objeto.nombre.

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

La *instanciación* de clases usa la notación de funciones.

Hacé de cuenta que el objeto de clase es una función sin parámetros que retorna una nueva instancia de la clase.

```
x = MyClass()
```

Muchas clases necesitan crear objetos con instancias en un estado inicial particular.

```
def __init__(self):
    self.data = []
```

Cuando una clase define un método __init__(), la instanciación de la clase automáticamente invoca a __init__() para la instancia recién creada.

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

2.3) Objetos instancia

La única operación que es entendida por los objetos instancia es la referencia de atributos.

Hay dos tipos de nombres de atributos válidos, atributos de datos y métodos.

Si x es la instancia de MiClase creada más arriba, el siguiente pedazo de código va a imprimir el valor 16, sin dejar ningún rastro:

```
x.counter = 1

while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

El otro tipo de atributo de instancia es el *método*. Un método es una función que «pertenece a» un objeto.

x.f no es la misma cosa que MiClase.f; dado que x.f es un *objeto método*, no un objeto función.

2.4) Objeto método

La llamada x.f() es exactamente equivalente a MiClase.f(x)

En general, llamar a un método con una lista de *n* argumentos es equivalente a llamar a la función correspondiente con una lista de argumentos que es creada insertando el objeto del método antes del primer argumento.

2.5) Variable de clase y de instancia

En general, las variables de instancia son para datos únicos de cada instancia y las variables de clase son para atributos y métodos compartidos por todas las instancias de la clase.

```
class Dog:
```

```
    kind = 'canine'         # class variable shared by all
instances

    def __init__(self, name):
        self.name = name    # instance variable unique to
each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

Los datos compartidos pueden tener efectos inesperados que involucren objetos mutable como ser listas y diccionarios.

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

El diseño correcto de esta clase sería usando una variable de instancia:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each
dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

3) Algunas observaciones

Si el mismo nombre de atributo aparece tanto en la instancia como en la clase, la búsqueda del atributo prioriza la instancia:

```
>>> class Warehouse:
        purpose = 'storage'
        region = 'west'

>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

A menudo, el primer argumento de un método se llama self (uno mismo). Esto no es nada más que una convención: el nombre self no significa nada en especial para Python.

Observá que, sin embargo, si no seguís la convención tu código puede resultar menos legible a otros programadores de Python.

No es necesario que el la definición de la función esté textualmente dentro de la definición de la clase:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Todo valor es un objeto, y por lo tanto tiene una *clase* (también llamado su *tipo*). Ésta se almacena como objeto.__class__.

4) Herencia

```
class DerivedClassName(BaseClassName):

    <statement-1>
    .
    .
    .
    <statement-N>
```

Las clases derivadas pueden redefinir métodos de su clase base.

Un método redefinido en una clase derivada puede de hecho querer extender en vez de simplemente reemplazar al método de la clase base con el mismo nombre.

Python tiene dos funciones integradas que funcionan con herencia:

- isinstance(obj, int)

Será True sólo si obj.__class__ es int o alguna clase derivada de int.

- issubclass(bool, int)

Es True ya que bool es una subclase de int.

4.1) Herencia múltiple

```
class DerivedClassName(Base1, Base2, Base3):

    <statement-1>
    .
    .
    .
    <statement-N>
```

Si un atributo no se encuentra en `ClaseDerivada`, se busca en `Base1`, luego (recursivamente) en las clases base de `Base1`, y sólo si no se encuentra allí se lo busca en `Base2`, y así sucesivamente.

5) Variables privadas

Las variables «privadas» de instancia, que no pueden accederse excepto desde dentro de un objeto, no existen en Python.

Sin embargo, hay una convención que se sigue en la mayoría del código Python: un nombre prefijado con un guión bajo (por ejemplo, _spam) debería tratarse como una parte no pública de la API (más allá de que sea una función, un método, o un dato).

Cualquier identificador con la forma __spam (al menos dos guiones bajos al principio, como mucho un guión bajo al final) es textualmente reemplazado por _nombredeclase__spam, donde nombredeclase es el nombre de clase actual.

6) Cambalache (Odds and Ends)

Una definición de clase vacía funcionará perfecto:

```
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Por ejemplo, si tenés una función que formatea algunos datos a partir de un objeto archivo, podés definir una clase con métodos read() y readline() que obtengan los datos de alguna cadena en memoria intermedia, y pasarlo como argumento.

Los objetos método de instancia tienen atributos también: m.__self__ es el objeto instancia con el método m(), y m.__func__ es el objeto función correspondiente al método.

7) Iteradores

En bambalinas, la sentencia for llama a iter() en el objeto contenedor. La función retorna un objeto iterador que define el método __next__() que accede elementos en el contenedor de a uno por vez. Cuando no hay más elementos, __next__() levanta una excepción StopIteration que le avisa al bucle del for que hay que terminar. Podés llamar al método __next__() usando la función integrada next().

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Agregando comportamiento de iterador a tus clases.

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

8) Generadores

Los generadores son una simple y poderosa herramienta para crear iteradores.

Están escritos como funciones regulares pero usan la sentencia **yield** cuando retornan datos.

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
```

```
...     print(char)
...
f
l
o
g
```

Todo lo que puede ser hecho con generadores también puede ser hecho con iteradores basados en clases, como se describe en la sección anterior. Lo que hace que los generadores sean tan compactos es que los métodos __iter__() y __next__() son creados automáticamente.

Cuando los generadores terminan automáticamente levantan StopIteration.

9) Expresiones generadoras

Algunos generadores simples pueden ser escritos de manera concisa como expresiones usando una sintaxis similar a las comprensiones de listas pero con paréntesis en lugar de corchetes.

Tienden a ser más amigables con la memoria que sus comprensiones de listas equivalentes.

```
>>> sum(i*i for i in range(10))             # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))      # dot product
260

>>> unique_words = set(word for line in page  for word in
line.split())

>>> valedictorian = max((student.gpa, student.name) for
student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

Más recursos sobre Python:

- https://www.python.org: El mayor sitio web de Python. Contiene código, documentación, y enlaces a páginas web relacionadas con Python. Esta web está replicada en varios sitios alrededor del mundo, como Europa, Japón y Australia; una réplica puede ser más rápida que el sitio principal, dependiendo de tu localización geográfica.

-https://docs.python.org: Acceso rápido a la documentación de Python.

-https://pypi.org: El Python Package Index, apodado previamente la Tienda de Queso 1, es un índice de módulos de Python creados por usuarios que están disponibles para su descarga. Cuando empiezas a distribuir código, lo puedes registrar allí para que otros lo encuentren.

- https://code.activestate.com/recipes/langs/python/: El Python Cookbook es una gran colección de ejemplos de código, módulos grandes y scripts útiles. Las contribuciones más notables también están recogidas en un libro titulado Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.)

- http://www.pyvideo.org recoge enlaces a vídeos relacionados con Python provenientes de conferencias y de reuniones de grupos de usuarios.

- https://scipy.org: El proyecto de Python científico incluye módulos para el cálculo rápido de operaciones y manipulaciones sobre arrays además de muchos paquetes para cosas como Álgebra Lineal, Transformadas de Fourier, solucionadores de sistemas no-lineales, distribuciones de números aleatorios, análisis estadísticos y otras herramientas.