Introducción a Python

Laboratorio de Instrumentación Virtual y Robótica Aplicada
UNMdP

Índice

1. Introducción

Python es un lenguaje de alto nivel, de fácil aprendizaje con un simple pero efectivo enfoque orientado a objetos. Cuenta con una sintaxis elegante con tipos dinámicos. El código es interpretado, es decir que pasa por un programa intérprete que lo ejecuta línea por línea.

A partir del intérprete podemos ejecutar instrucciones y obtener el resultado de forma inmediata o ejecutar un programa entero, almacenado con extensión `.py`. Al usar el intérprete, se indica con `>>>` que está esperando una instrucción. También podemos agregar comentarios usando `#` que hace que el resto de la línea a partir del `#` sea un comentario y se ignore para la ejecución.

2. Operaciones con números

Las operaciones numéricas se ingresan de la forma esperada, respetando el orden de operaciones. Los símbolos utilizados son:

- `+` para suma
- `-` para resta
- `*` multiplicación
- `/` división
- `//` división entera (descarta decimales)
- `%` resto de la división
- `**` potencia

Por ejemplo:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5
1.6
>>> 8 // 5
1
>>> 8 % 5
3
>>> 3**2
9
```

3. Variables

Una vez obtenidos los resultados, se pueden almacenar en una variable. No es necesario indicar el tipo de la variable como también se pueden almacenar diferentes tipos de valores dentro de una misma variable. Para esto se utiliza el símbolo =, como por ejemplo:

```
>>> a = 1 + 2
>>> b = a / 2
>>> b
1.5
>>> b = "texto"
>>> b
'texto'
```

También se puede combinar un operador con la asignación para modificar una variable:

```
>>> a = 1
>>> a += 1
>>> a
2
>>> a *= 2
>>> a
4
>>> a /= 2
>>> a
2.0
```

Como se observa en el ejemplo, al ingresar como instrucción el nombre de una variable, en el intérprete muestra el valor de la misma. Sin embargo, hay otra forma de mostrar información y es utilizando la instrucción `print()`:

```
>>> print(b)
texto
```

Es posible ingresar varios parámetros y se presentarán todos separados por espacios:

```
>>> print("cadena", 1, 2, b)
texto 1 2 texto
```

4. Cadenas de texto

Existen tres maneras de ingresar texto con Python:

- `"texto"`

- `'texto'`
- `"""texto
  multilínea"""`

Pueden utilizarse de forma indistinta pero en las cadenas de texto encerradas por comillas simples, pueden ingresarse comillas dobles y viceversa.

Existen cadenas de textos especiales que sirven para incluir variables dentro de ellas. Se denominan *f-strings* y se utilizan colocando una f delante de las comillas. Dentro de la cadena se pueden indicar el nombre de las variables, su formato y algunos otros parámetros como por ejemplo:

```
>>> pi = 3.14159
>>> print(f"La variable pi contiene el valor {pi}")
La variable pi contiene el valor 3.14159
>>> print(f"{pi=}")
pi=3.14159
>>> print(f"Ahora con menos decimales: {pi:.2f}")
Ahora con menos decimales: 3.14
```

Las cadenas de texto tienen métodos asociados (funciones pertenecientes a objetos) que permiten manipularlas y obtener información relevante sobre ella. Se puede encontrar en el manual de referencia un listado de todos esos métodos. A continuación se presentan algunos ejemplos útiles:

```
>>> texto = "abc def ghi jkl"
>>> texto.split(" ")
['abc', 'def', 'ghi', 'jkl']
>>> texto.find("def")
4
>>> texto[texto.find("def")]
'd'
>>> texto.replace("def", "abc")
'abc abc ghi jkl'
>>> texto
'abc def ghi jkl'
```

5. Listas

Son similares a lo que en otros lenguajes denominan *arrays* o vectores. Estas estructuras son dinámicas por lo que se pueden agregar o quitar elementos una vez creados. Para crear una lista se utilizan el par de corchetes `[]` con los elementos separados por comas:

```
>>> a = [1, 2, 3]
```

Para acceder a los elementos de una lista, se selecciona el índice comenzando por 0:

```
1
>>> a[0]
>>> a[1]
2
```

También se puede obtener una sublista a partir del rango de los índices indicando desde qué índice, hasta cuál y con qué paso. Estos valores pueden omitirse. El primero corresponde al primer índice de la lista y, en caso de omitirse, corresponde al primero de la lista. El segundo, es el último índice no inclusive de la lista que si se omite es el último de la lista. Por último el paso es en cuánto incrementa el índice que, al omitirlo, es 1. En resumen:

```
lista[inicio:fin:paso]
```

Si se omiten estos valores, por defecto se asume `inicio` como 0, `fin` como la cantidad de elementos de la lista y `paso` como 1. A continuación se muestran unos ejemplos de su uso:

```
>>> a[1:2]
[2]
>>> a[:2]
[1, 2]
>>> a[1:]
[2, 3]
>>> a[0:3]
[1, 2, 3]
>>> a[0:3:2]
[1, 3]
>>> a[::2]
[1, 3]
```

Para agregar un elemento a la lista, se pueden usar los métodos (propios de cada lista) `append()` para agregar al final e `insert()` para agregar en un índice determinado. También se puede utilizar la función `len()` para consultar la cantidad de elementos que contiene. Por último, se puede utilizar la palabra clave **`del`** para eliminar un elemento de la lista:

```
>>> a
[1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> len(a)
4
>>> a.insert(1, 100)
>>> a
[1, 100, 2, 3, 4]
>>> len(a)
5
>>> del a[1]
[1, 2, 3, 4]
```

6. Flujo de control

Para controlar la forma en la que se ejecuta una porción de código, podemos encontrar a la palabra clave **`if`** para ejecutar algo bajo ciertas condiciones y las palabras claves **`for`** y **`while`** para repetir un bloque de código.

Otros lenguajes utilizan símbolos como `\{\}` o `begin-end` para encerrar un bloque, sin embargo, Python requiere ubicarlos dentro de cierto nivel de indentación, es decir las instrucciones deben tener la **misma** cantidad de espacios al comienzo de la línea. Una forma de garantizarlo es usando la tecla `<Tab>` donde, según el editor, usa el caracter especial de tabulación o convierte automáticamente a una cantidad de espacios.

6.1. Condicionales

Se componen de la palabra clave **`if`** seguida de una condición que pueda evaluarse como verdadero (**`True`**) o falso (**`False`**) y dos puntos `:`. El bloque siguiente de código se evalúa siempre y cuando la condición sea verdadera:

```
>>> a = 1
>>> if a == 1:
        print("La condición es verdadera")
    print("Esto se ejecuta siempre")
La condición es verdadera
Esto se ejecuta siempre
```

Algunos operadores que se pueden utilizar para generar las condiciones son:

- `==`: compara igualdad
- `!=`: compara desigualdad

- >: compara si es mayor
- <: compara si es menor
- >=: compara si es mayor o igual
- <=: compara si es menor o igual
- `in`: verifica pertenencia en una lista, en una cadena de caracteres, etc.

Estos operadores producen un valor *booleano*:

```
>>> 1 == 2
False
>>> 1 != 2
True
>>> "a" in "abcdef"
True
>>> 4 in [1, 2]
False
```

Si la condición no es verdadera, es posible redireccionar el flujo a otro bloque usando las palabras claves **`else`** y **`elif`**. Ambas se utilizan para los casos en que la condición inicial es falsa pero, en la segunda, vuelve a evaluar una nueva condición como si fuera **`else`** seguido de **`if`**:

```
>>> a = 1
>>> if a == 1:
        print("a es 1")
    elif a > 1:
        print("a no es 1 pero es mayor a 1")
    else:
        print("a no es 1 ni es mayor a 1")
a es 1
```

6.2. Ciclos

Podemos distinguir dos tipos de bucles: **`for`** y **`while`**. El primero repite un bloque de código hasta agotar un conjunto de valores iterables, mientras que el segundo se repite hasta que se cumpla cierta condición. Para ambos casos, el ciclo puede ser interrumpido arbitrariamente usando la palabra clave **`break`**.

Para el caso del ciclo **`for`**, lo que se ingresa a continuación es una (o varias) variables que se asignan al comienzo de cada repetición seguido de la palabra clave `in` y un *iterable*. Esto último se refiere a algo que contenga o genere una cantidad limitada elementos de forma ordenada. Dos formas básicas de utilizarlo es con el iterable que genera la función `range()` y la otra es recorriendo los elementos de una lista. Por ejemplo:

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

Por otro lado, el ciclo **while** se repite hasta que la condición sea falsa:

```
>>> a = 0
>>> while a < 3:
        print(a)
        a += 1
0
1
2
```

Se puede generar un ciclo infinito colocando siempre el valor verdadero:

```
>>> while True:
        print("Se repite infinitamente")
Se repite infinitamente
Se repite infinitamente
Se repite infinitamente
Se repite infinitamente
Se repite infinitamente
...
```

En este caso, podemos interrumpir la ejecución del ciclo usando la combinación <Ctrl-C>.

6.3. Excepciones

Al producirse algún error durante la ejecución, Python corta la ejecución y muestra una *excepción* que permite dar una idea de dónde puede prevenir el error. Normalmente vienen asociadas a un tipo (**ValueError**, **IndexError**,

NameError, etc.) en relación al error producido. Sin embargo, es posible capturar estos errores utilizando las palabras claves **try** y **except** durante ejecución y realizar algo al respecto. Esto se puede usar para salvar el error y continuar con la ejecución. Por ejemplo, si queremos acceder a una variable que no existe:

```
>>> variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'variable' isn't defined
```

En este caso, se produce un **NameError** indicando que la variable no está definida. Si colocamos un bloque **try**, podemos capturar esta excepción:

```
>>> try:
        variable
    except:
        print("La variable no existe, le asignamos un valor")
        variable = 1
La variable no existe, le asignamos un valor
>>> variable
1
```

Es buena práctica luego de **except**, indicar el tipo de excepción a capturar. En caso de omitirlo, captura todas. En nuestro caso sería **NameError**. Esto es útil a la hora de capturar cierto tipo de errores como pueden ser que no exista un archivo, que no pueda conectarse a la red, etc.

7. Funciones

Para evitar repetir bloques de código se puede hacer uso de las funciones que pueden o no devolver un valor. Para definirlas, incluimos en cualquier parte del código la palabra **def** seguida del nombre de la función y sus parámetros:

```
>>> def funcion(a, b, c):
        print("Se llamó a la función con parámetros", a, b, c)
>>> funcion(1, 2, 3)
Se llamó a la función con parámetros 1 2 3
>>> funcion("a", 0, True)
Se llamó a la función con parámetros a 0 True
```

En este caso, la función no devuelve ningún valor, solo muestra información en pantalla. Por otro lado, podemos definir una función que realice una operación específica y devuelva un valor utilizando la palabra clave **return**:

```
>>> def suma_primeros_n_numeros(n):
        total = 0
        for i in range(1, n + 1):
            total += i

        return total
>>> suma_primeros_n_numeros(5)
15
```

8. Módulos

El lenguaje Python permite modularizar código de una forma muy sencilla. Es posible acceder a definiciones en otro archivo a partir de la palabra clave **`import`**. Supongamos que existe un archivo que se llama `modulo.py` en el directorio actual de trabajo con el siguiente contenido:

```
def opuesto(a):
    return -a

def saludo():
    print("Hola")
```

Podemos acceder a estas funciones de la siguiente manera:

```
>>> import modulo
>>> modulo.saludo()
Hola
>>> modulo.opuesto(1)
-1
```

Una forma alternativa de acceder a estas funciones es importándolas individualmente de la siguiente manera:

```
>>> from modulo import saludo
>>> saludo()
Hola
```