Tipos de datos, Variables, Operadores, Sentencias condicionales y de control

Mario González
Facultad de Ingeniería y Ciencias Aplicadas
Universidad de las Américas, Quito, Ecuador

Agosto, 2021

Valores y Tipos de Datos I

- Los valores son las cosas fundamentales que un programa manipula, ejemplo
  - La operación **1+1**, da como resultado **2**, que es un valor.
  - **“Hola, mundo”**, es otro valor.
- Estos valores tienen diferentes tipos de datos:
  - 2 es un **entero**,
  - “Hola, mundo” es una cadena de carácteres (**string**).

```
>>> type("Hola, mundo")
<type 'str'>
>>> size = 1
>>> type(size)
<type 'int'>
>>> size<0
False
>>> type(_)
<type 'bool'>
```

Valores y Tipos de Datos II

Python tiene muchos tipos de datos nativos. Estos son los más importantes:

1. **Booleanos:** son Verdadero o Falso.
2. **Números:** pueden ser números **enteros** (1 y 2), flotantes **(float)** (1.12 y 1.5), fracciones (1/2 y 2/3), o incluso números complejos.
3. **Cadenas de caracteres** Unicode, ejemplo un documento html.
4. **Bytes y matrices de bytes**, ejemplo un archivo de imagen JPEG.
5. **Listas:** ordenan secuencias de valores.
6. **Tuplas:** ordenan secuencias inmutables de valores.
7. **Conjuntos:** son colecciones desordenadas de valores.
8. **Los diccionarios:** son colecciones no ordenadas de pares clave-valor.

Variables I

- Una de las características más poderosas de un lenguaje de programación es la capacidad de manipular variables.
- **Una variable** es un nombre que hace referencia a un valor.
- La sentencia de asignación crea nuevas variables y les da valores:

```
<variable> = <expresion>
```

- Lea las instrucciones de asignación de derecha a izquierda:
  1. Evalúe la expresión de la derecha.
  2. Asigne la variable de la izquierda para referirse a ese valor.

```
>>> message = "What's up, Doc?"
>>> n = 17
>>> pi = 3.1415
```

Variables II

- Se puede asignar valores a una misma variable cuantas veces se quiera. El efecto es que la variable, en cada instante, sólo recuerda el último valor asignado... hasta que se le asigne otro.
- ¿Qué ocurre si hacemos referencia a una variable que no ha sido asignada?
- Los programadores generalmente eligen nombres para las variables que son significativas: documentan para qué se utiliza la variable.
- Los nombres de variables pueden ser arbitrariamente largos. Pueden contener tanto letras como números, pero tienen que comenzar con una letra.
- El uso de mayúsculas y minúsculas es importantes. Bruce y bruce son diferentes variables.

Variables III

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more$ = 1000000
SyntaxError: invalid syntax
>>> class = 'Computer Science 101'
SyntaxError: invalid syntax
```

- Resulta que **class** es una de las palabras clave de Python. Las palabras clave definen las reglas y la estructura del lenguaje, y no pueden ser utilizadas como nombres de variables.

```
and       def       exec      if        not       return
assert    del       finally   import    or        try
break     elif      for       in        pass      while
class     else      from      is        print     yield
continue  except    global    lambda    raise
```

Operadores y operandos I

- Los **operadores** son símbolos especiales que representan cálculos como la suma y la multiplicación.
- Los valores que el operador usa son llamados **operandos**.

```
20+32                    minute/60
hour-1                   5**2
hour*60+minute           (5+9)*(15-7)
```

- Los símbolos **+, -,** /, y el uso de paréntesis para agrupar, significan en Python lo que significan en matemática.
- El asterisco **(*)** es el símbolo de la multiplicación, y ** es el símbolo de la exponenciación.
- Cuando un nombre de una variable aparece en el lugar de un operando, se reemplaza con su valor antes de que se realiza la operación.
- Suma, resta, multiplicación y exponenciación todos hacen lo que se espera, pero puede ser sorprendido por la división.
- Orden de las operaciones:

Operadores y operandos II

- Los paréntesis tienen la prioridad más alta y se puede utilizar para forzar que una expresión sea evaluada en el orden que desee.
- Dado que las expresiones entre paréntesis se evalúan primero, `2 * (3-1)` es 4, y `(1 + 1) ** (5-2)` es 8.
- También puede utilizarse paréntesis para hacer una expresión más fácil de leer, como en `(minuto * 100) / 60`, a pesar de que no cambia el resultado.
- La Exponenciación tiene la siguiente precedencia más alta, por lo que `2 ** 1 + 1 es 3` y no 4, y `3 * 1 ** 3` es 3 y no 27.
- La multiplicación y la división tienen la misma prioridad, que es mayor que la suma y la resta, que también tienen la misma precedencia. Así que `2 * 3-1` produce 5 en lugar de 4, y `2 / 3-1` es -1, no 1 (recordar que en la división de enteros, `2/3 = 0`).
- Los operadores con la misma precedencia se evalúan de izquierda a derecha. Así, en la expresión `59*100/60`, la multiplicación ocurre primero, produciendo `5900/60`, lo que a su vez produce 98. Si las operaciones se habían evaluado de derecha a izquierda, el resultado habría sido `59 * 1`, que es 59, lo cual es incorrecto.

Operadores y operandos III

- Operaciones sobre cadenas de caracteres (**strings**)
  - En general, no se pueden realizar operaciones matemáticas sobre cadenas de caracteres, aún si las cadenas parecen números. Las siguientes expresiones son ilegales:

```
message-1  'Hello'/123  message*'Hello'  '15'+2
```

  - Para cadenas de caracteres, el operador + representa la concatenación, que significa unir los dos operandos uniéndolos de extremo a extremo. Por ejemplo:

```
fruit = 'banana'
bakedGood = ' nut bread'
print(fruit + bakedGood)
```

  - Así como `4 * 3` es equivalente a `4 + 4 + 4`, esperamos `'Fun' * 3` sea lo mismo que el `'Fun' + 'Fun' + 'Fun'`, y lo es. Por otro lado, hay una manera significativa en el que la concatenación de cadenas y la repetición son diferentes a la adición y multiplicación de enteros. ¿Pueden pensar en una propiedad que la suma y la multiplicación tienen y que la concatenación de cadenas y la repetición no?

Operadores y operandos IV

- **Los operadores de comparación** devuelven valores booleanos, verdadero (True) si se cumple la operación de comparación, y falso (False) en caso contrario.

operador	comparación
==	es igual que
!=	es distinto de
<	es menor que
<=	es menor o igual que
>	es mayor que
>=	es mayor o igual que

```
>>> 2<4
True
>>> 2>4
False
>>> 2==4
False
>>> 2==1+1
True
```

- Note que el operador de comparación == es diferente al operador de asignación =.

Operadores y operandos V

- Características de los operadores Python. El nivel de precedencia 1 es el de mayor prioridad.

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	**	Binario	Por la derecha	1
Identidad	+	Unario	—	2
Cambio de signo	-	Unario	—	2
Multiplicación	*	Binario	Por la izquierda	3
División	/	Binario	Por la izquierda	3
Módulo (o resto)	%	Binario	Por la izquierda	3
Suma	+	Binario	Por la izquierda	4
Resta	-	Binario	Por la izquierda	4
Igual que	==	Binario	—	5
Distinto de	!=	Binario	—	5
Menor que	<	Binario	—	5
Menor o igual que	<=	Binario	—	5
Mayor que	>	Binario	—	5
Mayor o Igual que	>=	Binario	—	5
Negación	**not**	Unario	—	6
Conjunción	**and**	Binario	Por la izquierda	7
Disyunción	**or**	Binario	Por la izquierda	8

Operadores y operandos VI

- Hay tres operadores lógicos en Python: la “y lógica”, o conjunción (and), la “o lógica” o disyunción (or) y el “no lógico” o negación (not).
- Estas son las tablas de verdad correspondientes:

and		
operandos		**resultado**
izquierdo	derecho	
True	*True*	*True*
True	*False*	*False*
False	*True*	*False*
False	*False*	*False*

or		
operandos		**resultado**
izquierdo	derecho	
True	*True*	*True*
True	*False*	*True*
False	*True*	*True*
False	*False*	*False*

not	
operando	**resultado**
True	*False*
False	*True*

Operadores y operandos VII

- Ejemplos:

```
>>> True and False
False
>>> not True
False
>>> (True and False) or True
True
>>> True and True or False
True
>>> False and True or True
True
>>> False and True or False
False
```

Operadores y operandos VIII

- Aridad, asociatividad y precedencia de los operadores lógicos

Operación	**Operador**	**Aridad**	**Asociatividad**	**Precedencia**
Negación	**not**	Unario	—	alta
Conjunción	**and**	Binario	Por la izquierda	media
Disyunción	**or**	Binario	Por la izquierda	baja

Composición

- Una de las características más útiles de los lenguajes de programación es su capacidad para tomar pequeños bloques de construcción y hacer composiciones complejas con ellos.

```
>>>  print(17 + 3)
20
print('Number of minutes: ', hour*60+minute)
percentage = (minute * 100) / 60
```

- Esta habilidad puede parecer poco impresionante, pero veremos otros ejemplos en los que la composición hace posible expresar cálculos complejos de forma clara y concisa.
- Hay límites en donde se puede utilizar ciertas expresiones. Por ejemplo, el lado izquierdo de una sentencia de asignación tiene que ser un nombre de variable, no una expresión. Así, lo siguiente es ilegal: `minuto + 1 = horas`.

Comentarios

- A medida que los programas se hacen más grandes y más complicados, se hacen más difícil de leer. Los lenguajes formales son densos, y con frecuencia es difícil de mirar un trozo de código y averiguar lo que está haciendo, o por qué.
- Por esta razón, es una buena idea añadir notas a los programas para explicar en lenguaje natural lo que el programa está haciendo. Estas notas se denominan comentarios, y están marcados con el símbolo #.

```
# Calcula el porcentaje de la hora transcurrido
percentage=(minute*100)/60
percentage=(minute*100)/60 # division entera
```

- Los comentarios son ignorados no tiene ningún efecto sobre el programa. El mensaje está dirigido al programador o para los programadores posibles usuarios de este código en el futuro.

```
""" This is an example of a multiline
comment that spans multiple lines
"""
```

Funciones predefinidas I

- Llamadas de funciones:

```
>>> betty = type("32")
>>> print(betty)
<type 'str'>
>>> id(3)
134882108
>>> betty = 3
>>> id(betty)
134882108
```

- Cada valor tiene un id, que es un número único relacionado con donde se almacena en la memoria del ordenador. El identificador de una variable es el id del valor al que se refiere.

Funciones predefinidas II

- Conversión de tipos de datos:

```
>>> int("32")
32
>>> int("Hello")
ValueError: invalid literal for int(): Hello
>>> int(3.99999)
3
>>> int(-2.3)
-2
>>> float(32)
32.0
>>> float("3.14159")
3.14159
>>> str(3.14149)
'3.14149'
```

Funciones predefinidas III

- Funciones matemáticas:
  - Python tiene un módulo de matemáticas que proporciona la mayor parte de las funciones matemáticas conocidas. Un módulo es un archivo que contiene una colección de funciones relacionadas.
  - Antes de poder utilizar las funciones de un módulo, tenemos que importarlas:

```
>>> import math
>>> decibel = math.log10 (17.0)
>>> angle = 1.5
>>> height = math.sin(angle)
>>> degrees = 45
>>> angle = degrees * 2 * math.pi / 360.0
>>> math.sin(angle)
0.707106781187
>>> math.sqrt(2) / 2.0
0.707106781187
```

Añadiendo funciones I

- La sintaxis para la definición de una función es:

```
def NAME( LIST OF PARAMETERS ):
  STATEMENTS
```

- Definición de una función:

```
def newLine():
  print()
```

- Uso de una función

```
print("First Line.")
newLine()
print("Second Line.")
```

Añadiendo funciones II

- Parámetros y argumentos:

```
def printTwice(x):
  print(x, x)
```

- Los argumentos son pasados a la función que son asignados a variables llamados parámetros:

```
>>> printTwice('Hola')
Hola Hola
```

Condicionales I

- Para escribir programas útiles, casi siempre necesitamos la capacidad de comprobar ciertas condiciones y cambiar el comportamiento del programa en consecuencia. Las sentencias condicionales nos dan esta capacidad. La forma más sencilla es la sentencia **if:**

```
if x > 0:
  print("x is positive")
```

```
HEADER:
  FIRST STATEMENT
  ...
  LAST STATEMENT
```

Condicionales II

- Ejecución alternativa:

```
if x%2 == 0:
  print(x, "es par")
else:
  print(x, "es impar")
```

- A veces hay más de dos posibilidades y necesitamos más de dos ramas. Una forma de expresar un cálculo de esa manera es usando condicionales encadenados:

```
if x < y:
  print(x, "is less than", y)
elif x > y:
  print(x, "is greater than", y)
else:
  print(x, "and", y, "are equal")
```

Condicionales III

- Las sentencias condicionales pueden ser anidadas dentro de otras:

```
if x == y:
  print(x, "and", y, "are equal")
else:
  if x < y:
    print(x, "is less than", y)
  else:
    print(x, "is greater than", y)
```

Condicionales IV

- La sentencia **return** permite terminar la ejecución de una función antes de llegar al final.

```
import math

def printLogarithm(x):
  if x <= 0:
    print("Positive numbers only, please.")
    return

  result = math.log(x)
  print("The log of x is", result)
```

Bucles (iteraciones) I

- La sentencia **while**:

```
count = 0
while (count < 9):
    print('The count is:', count)
    count = count + 1

print("Good bye!")
```

- Sentencia while con bucle infito:

```
var = 1
while var == 1 :  # Constructs an infinite loop
    num = input("Enter a number: ")
    print("You entered: ", num)

print("Good bye!")
```

Bucles (iteraciones) II

- La sentencia for en Python itera través de los elementos de cualquier secuencia (una lista o una cadena), en el orden en que aparecen en la secuencia.

```
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
```

- Podemos iterar también por una sección de la lista

```
words = ['cat', 'window', 'defenestrate']
for w in words[1:]:
    print(w, len(w))
```

Bucles (iteraciones) III

- La función **range()**:

```
for i in range(5):
    print(i)

for i in range(2,10):
    print(i)

a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print(i, a[i])
```

Bucles (iteraciones) IV

- Uso de las sentencias **break**, **else** y **continue**:

```
for n in range(2, 10):
  for x in range(2, n):
    if n % x == 0:
      print(n, 'equals', x, '*', n/x)
      break #termina el bucle
  else:
    # loop termina sin encontrar un factor
    print(n, 'es un numero primo')
```

  - La sentencia **break** rompe la envolvente más inmediata del for o del while
  - Sí, el código correcto. Visto de cerca: la cláusula **else** pertenece al bucle for, no la sentencia if.

Bucles (iteraciones) V

- La sentencia **continue**, continúa con la siguiente iteración del bucle:

```
for num in range(2, 10):
  if num % 2 == 0:
    print("Numero par encontrado", num)
    continue
  print("Numero encontrado", num)
```

Entrada por teclado

- Los programas que hemos escrito hasta ahora son un poco toscos en el sentido de que no aceptan ninguna entrada del usuario.
- Python proporciona funciones integradas que reciben entrada desde el teclado. El más simple es **input**.
- Antes de llamar **input**, es una buena idea imprimir un mensaje indicando al usuario qué entrada. Este mensaje se llama **prompt**. Podemos suministrar un **prompt** como argumento para **input**.

```
suma=0
v=int(input("No. de valores a sumar: "))
for i in range(0,v):
  x=float(input("Ingrese no. "+str(i+1)+": "))
  suma=suma+x

print("La suma es",suma)
```