Chapter 2

Variables, expresiones y sentencias

2.1 Valores y tipos

Un *valor* es una de las cosas básicas que utiliza un programa, como una letra o un número. Los valores que hemos visto hasta ahora han sido `1`, `2`, y "¡Hola, mundo!"

Esos valores pertenecen a *tipos* diferentes: `2` es un entero (int), y "¡Hola, mundo!" es una *cadena* (string), que recibe ese nombre porque contiene una "cadena" de letras. Tú (y el intérprete) podéis identificar las cadenas porque van encerradas entre comillas.

La sentencia `print` también funciona con enteros. Vamos a usar el comando `python` para iniciar el intérprete.

```
python
>>> print(4)
4
```

Si no estás seguro de qué tipo de valor estás manejando, el intérprete te lo puede decir.

```
>>> type(''¡Hola, mundo!)
<class 'str'>
>>> type(17)
<class 'int'>
```

Not surprisingly, strings belong to the type `str` and integers belong to the type `int`. Less obviously, numbers with a decimal point belong to a type called `float`, because these numbers are represented in a format called *floating point*.

```
>>> type(3.2)
<class 'float'>
```

¿Qué ocurre con valores como "17" y "3.2"? Parecen números, pero van entre comillas como las cadenas.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

Son cadenas.

Cuando escribes un entero grande, puede que te sientas tentado a usar comas o puntos para separarlo en grupos de tres dígitos, como en `1,000,000` [1]. Eso no es un entero válido en Python, pero en cambio sí que resulta válido algo como:

```
>>> print(1,000,000)
1 0 0
```

Bien, ha funcionado. ¡Pero eso no era lo que esperábamos!. Python interpreta `1,000,000` como una secuencia de enteros separados por comas, así que lo imprime con espacios en medio.

Éste es el primer ejemplo que hemos visto de un error semántico: el código funciona sin producir ningún mensaje de error, pero no hace su trabajo "correctamente".

2.2 Variables

Una de las características más potentes de un lenguaje de programación es la capacidad de manipular *variables*. Una variable es un nombre que se refiere a un valor.

Una *sentencia de asignación* crea variables nuevas y las da valores:

```
>>> mensaje = 'Y ahora algo completamente diferente'
>>> n = 17
>>> pi = 3.1415926535897931
```

Este ejemplo hace tres asignaciones. La primera asigna una cadena a una variable nueva llamada `mensaje`; la segunda asigna el entero `17` a `n`; la tercera asigna el valor (aproximado) de π a `pi`.

Para mostrar el valor de una variable, se puede usar la sentencia print:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

El tipo de una variable es el tipo del valor al que se refiere.

[1] En el mundo anglosajón el "separador de millares" es la coma, y no el punto (Nota del trad.)

```
>>> type(mensaje)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

2.3 Nombres de variables y palabras claves

Los programadores generalmente eligen nombres para sus variables que tengan sentido y documenten para qué se usa esa variable.

Los nombres de las variables pueden ser arbitrariamente largos. Pueden contener tanto letras como números, pero no pueden comenzar con un número. Se pueden usar letras mayúsculas, pero es buena idea comenzar los nombres de las variables con una letras minúscula (veremos por qué más adelante).

El carácter guión-bajo (_) puede utilizarse en un nombre. A menudo se utiliza en nombres con múltiples palabras, como en `mi_nombre` o `velocidad_de_golondrina_sin_carga`. Los nombres de las variables pueden comenzar con un carácter guión-bajo, pero generalmente se evita usarlo así a menos que se esté escribiendo código para librerías que luego utilizarán otros.

Si se le da a una variable un nombre no permitido, se obtiene un error de sintaxis:

```
>>> 76trombones = 'gran desfile'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Teorema avanzado de Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` es incorrecto porque comienza por un número. `more@` es incorrecto porque contiene un carácter no premitido, `@`. Pero, ¿qué es lo que está mal en `class`?

Pues resulta que `class` es una de las *palabras clave* de Python. El intérprete usa palabras clave para reconocer la estructura del programa, y esas palabras no pueden ser utilizadas como nombres de variables.

Python reserva 33 palabras claves para su propio uso:

```
and       del       from      None      True
as        elif      global    nonlocal  try
assert    else      if        not       while
break     except    import    or        with
class     False     in        pass      yield
continue  finally   is        raise
def       for       lambda    return
```

Puede que quieras tener esta lista a mano. Si el intérprete se queja por el nombre de una de tus variables y no sabes por qué, comprueba si ese nombre está en esta lista.

2.4 Sentencias

Una *sentencia* es una unidad de código que el intérprete de Python puede ejecutar. Hemos visto hasta ahora dos tipos de sentencia: print y las asignaciones.

Cuando escribes una sentencia en modo interactivo, el intérprete la ejecuta y muestra el resultado, si es que lo hay.

Un script normalmente contiene una secuencia de sentencias. Si hay más de una sentencia, los resultados aparecen de uno en uno según se van ejecutando las sentencias.

Por ejemplo, el script

```
print(1)
x = 2
print(x)
```

produce la salida

```
1
2
```

La sentencia de asignación no produce ninguna salida.

2.5 Operadores y operandos

Los operadores son símbolos especiales que representan cálculos, como la suma o la multiplicación. Los valores a los cuales se aplican esos operadores reciben el nombre de *operandos*.

Los operadores `+`, `-`, , `/`, y `\*` realizan sumas, restas, multiplicaciones, divisiones y exponenciación (elevar un número a una potencia), como se muestra en los ejemplos siguientes:

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

Ha habido un cambio en el operador de división entre Python 2.x y Python 3.x. En Python 3.x, el resultado de esta división es un resultado de punto flotante:

```
>>> minute = 59
>>> minute/60
0.9833333333333333
```

El operador de división en Python 2.0 dividiría dos enteros y truncar el resultado a un entero:

```
>>> minute = 59
>>> minute/60
0
```

Para obtener la misma respuesta en Python 3.0 use división dividida (// integer).

```
>>> minute = 59
>>> minute//60
0
```

En Python 3, la división de enteros funciona mucho más como cabría esperar. Si ingresaste la expresión en una calculadora.

2.6 Expresiones

Una *expresión* es una combinación de valores, variables y operadores. Un valor por si mismo se considera una expresión, y también lo es una variable, así que las siguientes expresiones son todas válidas (asumiendo que la variable `x` tenga un valor asignado):

```
17
x
x + 17
```

Si escribes una expresión en modo interactivo, el intérprete la *evalúa* y muestra el resultado:

```
>>> 1 + 1
2
```

Sin embargo, en un script, ¡una expresión por si misma no hace nada! Esto a menudo puede producir confusión entre los principiantes.

Ejercicio 1: Escribe las siguientes sentencias en el intérprete de Python para comprobar qué hacen:

```
5
x = 5
x + 1
```

2.7 Orden de las operaciones

Cuando en una expresión aparece más de un operador, el orden de evaluación depende de las *reglas de precedencia*. Para los operadores matemáticos, Python sigue las convenciones matemáticas. El acrónimo *PEMDSR* resulta útil para recordar esas reglas:

- Los *P*aréntesis tienen el nivel superior de precedencia, y pueden usarse para forzar a que una expresión sea evaluada en el orden que se quiera. Dado que las expresiones entre paréntesis son evaluadas primero, `2 * (3-1)` es 4, y `(1+1)**(5-2)` es 8. Se pueden usar también paréntesis para hacer una expresión más sencilla de leer, incluso si el resultado de la misma no varía por ello, como en `(minuto * 100) / 60`.
- La *E*xponenciación (elevar un número a una potencia) tiene el siguiente nivel más alto de precedencia, de modo que `2**1+1` es 3, no 4, y `3*1**3` es 3, no 27.
- La *M*ultiplicación y la *D*ivisión tienen la misma precedencia, que es superior a la de la *S*uma y la *R*esta, que también tienen entre si el mismo nivel de precedencia. Así que `2*3-1` es 5, no 4, y `6+4/2` es 8, no 5.
- Los operadores con igual precedencia son evaluados de izquierda a derecha. Así que la expresión `5-3-1` es 1 y no 3, ya que `5-3` se evalúa antes, y después se resta `1` de `2`.

En caso de duda, añade siempre paréntesis a tus expresiones para asegurarte de que las operaciones se realizan en el orden que tú quieres.

2.8 Operador módulo

El *operador módulo* trabaja con enteros y obtiene el resto de la operación consistente en dividir el primer operando por el segundo. En Python, el operador módulo es un signo de porcentaje (`%`). La sintaxis es la misma que se usa para los demás operadores:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

Así que 7 dividido por 3 es 2 y nos sobra 1.

El operador módulo resulta ser sorprendentemente útil. Por ejemplo, puedes comprobar si un número es divisible por otro—si `x % y` es cero, entonces `x` es divisible por `y`.

También se puede extraer el dígito más a la derecha de los que componen un número. Por ejemplo, `x % 10` obtiene el dígito que está más a la derecha de `x` (en base 10). De forma similar, `x % 100` obtiene los dos últimos dígitos.

2.9 Operaciones con cadenas

El operador + funciona con las cadenas, pero no realiza una suma en el sentido matemático. En vez de eso, realiza una *concatenación*, que quiere decir que une ambas cadenas, enlazando el final de la primera con el principio de la segunda. Por ejemplo:

```
>>> primero = 10
>>> segundo = 15
>>> print(primero+segundo)
25
>>> primero = '100'
>>> segundo = '150'
>>> print(primero + segundo)
100150
```

La salida de este programa es `100150`.

El operador * también trabaja con cadenas multiplicando el contenido de una cadena por un entero. Por ejemplo:

```
>>> primero = 'Test '
>>> second = 3
>>> print(primero * second)
Test Test Test
```

2.10 Petición de información al usuario

A veces necesitaremos que sea el usuario quien nos proporcione el valor para una variable, a través del teclado. Python proporciona una función interna llamada `input` que recibe la entrada desde el teclado. Cuando se llama a esa función, el programa se detiene y espera a que el usuario escriba algo. Cuando el usuario pulsa `Retorno` o `Intro`, el programa continúa y `input` devuelve como una cadena aquello que el usuario escribió.

```
>>> entrada = input()
Cualquier cosa ridícula
>>> print(entrada)
Cualquier cosa ridícula
```

Antes de recibir cualquier dato desde el usuario, es buena idea escribir un mensaje explicándole qué debe introducir. Se puede pasar una cadena a `input`, que será mostrada al usuario antes de que el programa se detenga para recibir su entrada:

```
>>> nombre = input('¿Cómo te llamas?\n')
¿Cómo te llamas?
Chuck
>>> print(nombre)
Chuck
```

La secuencia `\n` al final del mensaje representa un *newline*, que es un carácter especial que provoca un salto de línea. Por eso la entrada del usuario aparece debajo de nuestro mensaje.

Si esperas que el usuario escriba un entero, puedes intentar convertir el valor de retorno a `int` usando la función `int()`:

```
>>> prompt = '¿Cual es la velocidad de vuelo de una golondrina sin carga?\n'
>>> velocidad = input(prompt)
¿Cual es la velocidad de vuelo de una golondrina sin carga?
17
>>> int(velocidad)
17
>>> int(velocidad) + 5
22
```

Pero si el usuario escribe algo que no sea una cadena de dígitos, obtendrás un error:

```
>>> velocidad = input(prompt)
¿Cual.... es la velocidad de vuelo de una golondrina sin carga?
¿Te refieres a una golondrina africana o a una europea?
>>> int(velocidad)
ValueError: invalid literal for int()
```

Veremos cómo controlar este tipo de errores más adelante.

2.11 Comentarios

A medida que los programas se van volviendo más grandes y complicados, se vuelven más difíciles de leer. Los lenguajes formales son densos, y a menudo es complicado mirar un trozo de código e imaginarse qué es lo que hace, o por qué.

Por eso es buena idea añadir notas a tus programas, para explicar en un lenguaje normal qué es lo que el programa está haciendo. Estas notas reciben el nombre de *comentarios*, y en Python comienzan con el símbolo `#`:

```
# calcula el porcentaje de hora transcurrido
porcentaje = (minuto * 100) / 60
```

En este caso, el comentario aparece como una línea completa. Pero también puedes poner comentarios al final de una línea

```
porcentaje = (minuto * 100) / 60      # porcentaje de una hora
```

Todo lo que va desde `#` hasta el final de la línea es ignorado—no afecta para nada al programa.

Las comentarios son más útiles cuando documentan características del código que no resultan obvias. Es razonable asumir que el lector puede descifrar *qué* es lo que el código hace; es mucho más útil explicarle *por qué*.

Este comentario es redundante con el código e inútil:

```
v = 5      # asigna 5 a v
```

Este comentario contiene información útil que no está en el código:

```
v = 5      # velocidad en metros/segundo.
```

Elegir nombres adecuados para las variables puede reducir la necesidad de comentarios, pero los nombres largos también pueden ocasionar que las expresiones complejas sean difíciles de leer, así que hay que conseguir una solución de compromiso.

2.12 Elección de nombres de variables mnemónicos

Mientras sigas las sencillas reglas de nombrado de variables y evites las palabras reservadas, dispondrás de una gran variedad de opciones para poner nombres a tus variables. Al principio, esa diversidad puede llegar a resultarte confusa, tanto al leer un programa como al escribir el tuyo propio. Por ejemplo, los tres programas siguientes son idénticos en cuanto a la función que realizan, pero muy diferentes cuando los lees e intentas entenderlos.

```
a = 35.0
b = 12.50
c = a * b
print(c)
```

```
horas = 35.0
tarifa = 12.50
salario = horas * tarifa
print(salario)
```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

El intérprete de Python ve los tres programas como *exactamente idénticos*, pero los humanos ven y asimilan estos programas de forma bastante diferente. Los humanos entenderán más rápidamente el *objetivo* del segundo programa, ya que el programador ha elegido nombres de variables que reflejan lo que pretendía de acuerdo al contenido que iba almacenar en cada variable.

Esa sabia elección de nombres de variables se denomina utilizar "nombres de variables mnemónicos". La palabra *mnemónico*[2] significa "que ayuda a memorizar".

[2]Consulta https://es.wikipedia.org/wiki/Mnemonico para obtener una descripción detallada de la palabra "mnemónico".

Elegimos nombres de variables mnemónicos para ayudarnos a recordar por qué creamos las variables al principio.

A pesar de que todo esto parezca estupendo, y de que sea una idea muy buena usar nombres de variables mnemónicos, ese tipo de nombres pueden interponerse en el camino de los programadores novatos a la hora de analizar y comprender el código. Esto se debe a que los programadores principiantes no han memorizado aún las palabras reservadas (sólo hay 33), y a veces variables con nombres que son demasiado descriptivos pueden llegar a parecerles parte del lenguaje y no simplemente nombres de variable bien elegidos[3].

Echa un vistazo rápido al siguiente código de ejemplo en Python, que se mueve en bucle a través de un conjunto de datos. Trataremos los bucles pronto, pero por ahora tan sólo trata de entender su significado:

```
for word in words:
    print(word)
```

¿Qué ocurre aquí? ¿Cuáles de las piezas (for, word, in, etc.) son palabras reservadas y cuáles son simplemente nombres de variables? ¿Acaso Python comprende de un modo básico la noción de palabras (`words`)? Los programadores novatos tienen problemas separando qué parte del código *debe* mantenerse tal como está en este ejemplo y qué partes son simplemente elección del programador.

El código siguiente es equivalente al de arriba:

```
for slice in pizza:
    print(slice)
```

Para los principiantes es más fácil estudiar este código y saber qué partes son palabras reservadas definidas por Python y qué partes son simplemente nombres de variables elegidas por el programador. Está bastante claro que Python no entiende nada de pizza ni de porciones, ni del hecho de que una pizza consiste en un conjunto de una o más porciones.

Pero si nuestro programa lo que realmente va a hacer es leer datos y buscar palabras en ellos, `pizza` y `porción` son nombres muy poco mnemónicos. Elegirlos como nombres de variables distrae del propósito real del programa.

Dentro de muy poco tiempo, conocerás las palabras reservadas más comunes, y empezarás a ver cómo esas palabras reservadas resaltan sobre las demás:

Las partes del código que están definidas por Python (`for`, `in`, `print`, y `:`) están en negrita, mientras que las variables elegidas por el programador (`word` y `words`) no lo están. Muchos editores de texto son conscientes de la sintaxis de Python y colorearán las palabras reservadas de forma diferente para darte pistas que te permitan mantener tus variables y las palabras reservadas separados. Dentro de poco empezarás a leer Python y podrás determinar rápidamente qué es una variable y qué es una palabra reservada.

[3] El párrafo anterior se refiere más bien a quienes eligen nombres de variables en inglés, ya que todas las palabras reservadas de Python coinciden con palabras propias de ese idioma (Nota del trad.)

2.13 Depuración

En este punto, el error de sintaxis que es más probable que cometas será intentar utilizar nombres de variables no válidos, como `class` y `yield`, que son palabras clave, o `odd~job` y `US$`, que contienen caracteres no válidos.

Si pones un espacio en un nombre de variable, Python cree que se trata de dos operandos sin ningún operador:

```
>>> bad name = 5
SyntaxError: invalid syntax

>>> month = 09
  File "<stdin>", line 1
    month = 09
            ^
SyntaxError: invalid token
```

Para la mayoría de errores de sintaxis, los mensajes de error no ayudan mucho. Los mensajes más comunes son `SyntaxError: invalid syntax` y `SyntaxError: invalid token`, ninguno de los cuales resulta muy informativo.

El runtime error (error en tiempo de ejecución) que es más probable que obtengas es un "use before def" (uso antes de definir); que significa que estás intentando usar una variable antes de que le hayas asignado un valor. Eso puede ocurrir si escribes mal el nombre de la variable:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Los nombres de las variables son sensibles a mayúsculas, así que `LaTeX` no es lo mismo que `latex`.

En este punto, la causa más probable de un error semántico es el orden de las operaciones. Por ejemplo, para evaluar $\frac{1}{2\pi}$, puedes sentirte tentado a escribir

```
>>> 1.0 / 2.0 * pi
```

Pero la división se evalúa antes, ¡así que obtendrás $\pi/2$, que no es lo mismo! No hay forma de que Python sepa qué es lo que querías escribir exactamente, así que en este caso no obtienes un mensaje de error; simplemente obtienes una respuesta incorrecta.

2.14 Glosario

asignación Una sentencia que asigna un valor a una variable.

cadena Un tipo que representa secuencias de caracteres.

concatenar Unir dos operandos, uno a continuación del otro.

comentario Información en un programa que se pone para otros programadores (o para cualquiera que lea el código fuente), y no tiene efecto alguno en la ejecución del programa.

división entera La operación que divide dos números y trunca la parte fraccionaria.

entero Un tipo que representa números enteros.

evaluar Simplificar una expresión realizando las operaciones en orden para obtener un único valor.

expresión Una combinación de variables, operadores y valores que representan un único valor resultante.

mnemónico Una ayuda para memorizar. A menudo damos nombres mnemónicos a las variables para ayudarnos a recordar qué está almacenado en ellas.

palabra clave Una palabra reservada que es usada por el compilador para analizar un programa; no se pueden usar palabres clave como `if`, `def`, y `while` como nombres de variables.

punto flotante Un tipo que representa números con parte decimal.

operador Un símbolo especial que representa un cálculo simple, como suma, multiplicación o concatenación de cadenas.

operador módulo Un operador, representado por un signo de porcentaje (`%`), que funciona con enteros y obtiene el resto cuando un número es dividido por otro.

operando Uno de los valores con los cuales un operador opera.

reglas de precedencia El conjunto de reglas que gobierna el orden en el cual son evaluadas las expresiones que involucran a múltiples operadores.

sentencia Una sección del código que representa un comando o acción. Hasta ahora, las únicas sentencias que hemos visto son asignaciones y sentencias print.

tipo Una categoría de valores. Los tipos que hemos visto hasta ahora son enteros (tipo `int`), números en punto flotante (tipo `float`), y cadenas (tipo `str`).

valor Una de las unidades básicas de datos, como un número o una cadena, que un programa manipula.

variable Un nombre que hace referencia a un valor.

2.15 Ejercicios

Ejercicio 2: Escribe un programa que use `input` para pedirle al usuario su nombre y luego darle la bienvenida.

```
Introduzca tu nombre: Chuck
Hola, Chuck
```

Ejercicio 3: Escribe un programa para pedirle al usuario el número de horas y la tarifa por hora para calcular el salario bruto.

```
Introduzca Horas: 35
Introduzca Tarifa: 2.75
Salario: 96.25
```

Por ahora no es necesario preocuparse de que nuestro salario tenga exactamente dos dígitos después del punto decimal. Si quieres, puedes probar la función interna de Python `round` para redondear de forma adecuada el salario resultante a dos dígitos decimales.

Ejercicio 4: Asume que ejecutamos las siguientes sentencias de asignación:

```
ancho = 17
alto = 12.0
```

Para cada una de las expresiones siguientes, escribe el valor de la expresión y el tipo (del valor de la expresión).

1. `ancho/2`
2. `ancho/2.0`
3. `alto/3`
4. `1 + 2 * 5`

Usa el intérprete de Python para comprobar tus respuestas.

Ejercicio 5: Escribe un programa que le pida al usuario una temperatura en grados Celsius, la convierta a grados Fahrenheit e imprima por pantalla la temperatura convertida.