Think Python
How to Think Like a Computer Scientist
(Traducción al español)

Allen Downey

Traducción: Brayan Calderon

Capítulo 1

El camino del programa

El objetivo de este libro es enseñarte a pensar como un **científico de la computación**. Esta forma de pensar combina algunas de las mejores características de las matemáticas, la ingeniería y las ciencias naturales. Como los matemáticos, los científicos de la computación usan lenguajes formales para denotar ideas (específicamente, computaciones). Como los ingenieros, diseñan cosas ensamblando componentes en sistemas y evaluando compromisos entre alternativas. Como los científicos, observan el comportamiento de sistemas complejos, forman hipótesis y prueban predicciones.

La habilidad más importante para un científico de la computación es la **resolución de problemas**. Resolver problemas significa tener la capacidad de formular problemas, pensar creativamente sobre soluciones y expresar una solución de forma clara y precisa. Como verás, el proceso de aprender a programar es una excelente oportunidad para practicar habilidades de resolución de problemas. Por eso, este capítulo se llama *"El camino del programa"*.

En un nivel, estarás aprendiendo a programar, una habilidad útil por sí misma. En otro nivel, estarás usando la programación como un medio para alcanzar otros fines. A medida que avancemos, ese propósito se volverá más claro.

1.1. ¿Qué es un programa?

Un programa es una secuencia de instrucciones que especifican cómo ejecutar una computación. La computación puede ser algo matemático, como solucionar un sistema de ecuaciones o determinar las raíces de un polinomio, pero también puede ser una computación simbólica, como buscar y reemplazar el texto de un documento o (aunque parezca raro) compilar un programa. Las instrucciones (comandos, órdenes) tienen una apariencia diferente en lenguajes de programación diferentes, pero existen algunas funciones básicas que se presentan en casi todo lenguaje:

Entrada: Recibir datos del teclado, o un archivo u otro aparato.

Salida: Mostrar datos en el monitor o enviar datos a un archivo u otro aparato.

Matemáticas: Ejecutar operaciones básicas de matemáticas como la adición y la multiplicación.

Operación Condicional: Probar la veracidad de alguna condición y ejecutar una secuencia de instrucciones apropiada.

Repetición: Ejecutar alguna acción repetidas veces, normalmente con alguna variación.

Lo crea o no, eso es todo. Todos los programas que existen, por complicados que sean, están formulados exclusivamente con tales instrucciones. Así, una manera de describir la programación es: El proceso de romper una tarea en tareas cada vez más pequeñas hasta que estas tareas sean suficientemente simples para ser ejecutadas con una de estas instrucciones simples. Quizás esta descripción sea un poco ambigua. No se preocupe. Lo explicaremos con más detalle con el tema de los algoritmos.

1.2. Ejecutando Python

Uno de los desafíos de empezar con Python es que podrías tener que instalar Python y software relacionado en tu computadora. Si estás familiarizado con tu sistema operativo, y especialmente si te sientes cómodo con la interfaz de línea de comandos, no tendrás problemas para instalar Python. Pero para principiantes, puede ser doloroso aprender sobre administración de sistemas y programación al mismo tiempo.

Para evitar ese problema, recomiendo que comiences ejecutando Python en un navegador. Más adelante, cuando te sientas cómodo con Python, haré sugerencias para instalar Python en tu computadora. Hay varias páginas web que puedes usar para ejecutar Python. Si ya tienes una favorita, adelante y úsala. De lo contrario, recomiendo PythonAnywhere. Proporciono instrucciones detalladas para empezar en http://tinyurl.com/thinkpython2e.

Hay dos versiones de Python, llamadas Python 2 y Python 3. Son muy similares, así que si aprendes una, es fácil cambiar a la otra. De hecho, solo hay unas pocas diferencias con las que te encontrarás como principiante. Este libro está escrito para Python 3, pero incluyo algunas notas sobre Python 2. El intérprete de Python es un programa que lee y ejecuta código Python. Dependiendo de tu entorno, podrías iniciar el intérprete haciendo clic en un icono, o escribiendo python en una línea de comandos. Cuando se inicia, deberías ver una salida como esta:

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more
   information.
```

Las primeras tres líneas contienen información sobre el intérprete y el sistema operativo en el que se está ejecutando, así que podría ser diferente para ti. Pero deberías verificar que el número de versión, que es 3.4.0 en este ejemplo, comience con 3, lo cual indica que estás ejecutando Python 3. Si comienza con 2, estás ejecutando (lo adivinaste) Python 2. La última línea es un prompt que indica que el intérprete está listo para que ingreses código. Si escribes una línea de código y presionas Enter, el intérprete muestra el resultado:

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

Ahora estás listo para comenzar. De aquí en adelante, asumo que sabes cómo iniciar el intérprete de Python y ejecutar código.

1.3. El primer programa

Tradicionalmente el primer programa en un lenguaje nuevo se llama "Hola, mundo" (Hello world!) porque sólo muestra las palabras "Hola, mundo". En Python es así:

```
>>> print('Hola, mundo')
```

Este es un ejemplo de una sentencia print, la cual no imprime nada en papel, más bien muestra un valor. En este caso, el resultado es las palabras:

```
Hola, mundo
```

Las comillas en el programa indican el inicio y el final del texto que se va a mostrar; no aparecen en el resultado. Los paréntesis indican que print es una función. Llegaremos a las funciones en el capítulo 3. En Python 2, las declaracions print es ligeremente diferente; no es una función, por lo que no usa paréntesis.

```
>>> print 'Hola, mundo'
```

Esta distición tendrá más sentido pronto, pero eso es suficiente para comenzar.

1.4. Operaciones Aritméticas

Después de "Hola, mundo", el siguiente paso es la aritmética. Python proporciona operadores, que son simbolos especiales que representan cálculos como sumas y multiplicación. Los operadores +, -, y * realizan suma, resta y multiplicación, como en los siguientes ejemplos:

```
>>> 40 + 2
42
>>> 43 - 1
```

```
42
>>> 6 * 7
42
```

El operador / realiza división:

```
>>> 84 / 2
42.0
```

Te podrías preguntar por qué el resultado es 42.0 en lugar de 42. Lo explicare en la sigueinte sección. Finalmente, el operador ** realiza exponenciación:

```
>>> 6 ** 2 + 6
42
```

En algunos lenguajes. ŝe usa para exponenciación, pero en Python es un operador a nivel de bits llamado XOR. Si no estás familiarizado con los operadores a nivel de bits, el resultado te sorprenderá:

```
>>> 6 ^ 2
4
```

No cubriré los operadores a nivel de bits en este libro, pero puedes leer sobre ellos en http://wiki.python.org/moin/BitwiseOperators

1.5. Valores y tipos

Un valor es una de las cosas básicas con las que trabaja un programa, como una letra o un número. Algunos valores que hemos visto hasta ahora son 2, 42.0, y 'Hola, Mundo!'.

Estos valores pertenecen a diferentes tipos: 2 es un entero, 42.0 es un número de punto flotante, y 'Hola, Mundo!' es una cadena, llamada así porque las letras que contiene están encadenadas juntas.

Si no estás seguro de qué tipo tiene un valor, el intérprete puede decírtelo:

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hola, Mundo!')
<class 'str'>
```

En estos resultados, la palabra "class" se usa en el sentido de una categoría; un tipo es una categoría de valores.

No es sorprendente que los enteros pertenezcan al tipo `int`, las cadenas pertenezcan a `str` y los números de punto flotante pertenezcan a `float`.

¿Qué pasa con valores como '2' y '42.0'? Parecen números, pero están entre comillas como las cadenas.

```
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
```

Son cadenas.

Cuando escribes un entero grande, podrías sentirte tentado a usar comas entre grupos de dígitos, como en 1,000,000. Esto no es un entero legal en Python, pero es legal:

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

¡Eso no es para nada lo que esperábamos! Python interpreta 1,000,000 como una secuencia de enteros separados por comas. Aprenderemos más sobre este tipo de secuencia más adelante.

1.6. Lenguajes formales y lenguajes naturales

Los **lenguajes naturales** son los lenguajes hablados por seres humanos, como el español, el inglés y el francés. No los han diseñado personas (aunque se intente poner cierto orden en ellos), sino que se han desarrollado naturalmente.

Los **lenguajes formales** son lenguajes diseñados por humanos y que tienen aplicaciones específicas. La notación matemática, por ejemplo, es un lenguaje formal ya que se presta a la representación de las relaciones entre números y símbolos. Los químicos utilizan un lenguaje formal para representar la estructura química de las moléculas. Y lo más importante:

> **Los lenguajes de programación son lenguajes formales desarrollados para expresar computaciones.**

Los lenguajes formales tienden a tener reglas de sintaxis estrictas que gobiernan la estructura de las declaraciones. Por ejemplo, en matemáticas la declaración $3 + 3 = 6$ tiene sintaxis correcta, pero $3+ = 3\$6$ no la tiene. En química H_2O es una fórmula sintácticamente correcta, pero ${}_2Zz$ no lo es.

Las reglas de sintaxis vienen en dos variedades, relacionadas con *tokens* y estructura. Los tokens son los elementos básicos del lenguaje, como palabras, números y elementos químicos. Uno de los problemas con $3+ = 3\$6$ es que \$ no es un token legal en matemáticas

(al menos que yo sepa). De manera similar, 2Zz no es legal porque no hay elemento con la abreviación Zz.

El segundo tipo de regla de sintaxis se refiere a la manera en que se combinan los tokens. La ecuación 3 + /3 es ilegal porque aunque + y / son tokens legales, no puedes tener uno justo después del otro. De manera similar, en una fórmula química el subíndice viene después del nombre del elemento, no antes.

Esta es @ una oración bien estructurada en espan̈*l con t*kens inválidos en ella. Esta oración todos tokens válidos tiene, pero estructura inválida con.

Cuando lees una oración en español o una declaración en un lenguaje formal, tienes que descifrar la estructura (aunque en un lenguaje natural haces esto subconscientemente). Este proceso se llama *análisis sintáctico*.

Aunque los lenguajes formales y naturales tienen muchas características en común —tokens, estructura y sintaxis— son algunas diferencias:

ambigüedad: Los lenguajes naturales están llenos de ambigüedad, con la cual las personas lidian usando pistas contextuales y otra información. Los lenguajes formales están diseñados para ser casi o completamente no ambiguos, lo cual significa que cualquier declaración tiene exactamente un significado, independientemente del contexto.

redundancia: Para compensar la ambigüedad y reducir malentendidos, los lenguajes naturales emplean mucha redundancia. Como resultado, a menudo son verbosos. Los lenguajes formales son menos redundantes y más concisos.

literalidad: Los lenguajes naturales están llenos de modismos y metáforas. Si digo "Se me cayó la ficha", probablemente no hay ficha y nada se está cayendo (este modismo significa que alguien entendió algo después de un período de confusión). Los lenguajes formales significan exactamente lo que dicen.

Porque todos crecemos hablando lenguajes naturales, a veces es difícil ajustarse a los lenguajes formales. La diferencia entre lenguaje formal y natural es como la diferencia entre poesía y prosa, pero más acentuada:

Poesía: Las palabras se usan tanto por sus sonidos como por su significado, y todo el poema junto crea un efecto o respuesta emocional. La ambigüedad no solo es común sino a menudo deliberada.

Prosa: El significado literal de las palabras es más importante, y la estructura contribuye más significado. La prosa es más susceptible al análisis que la poesía pero aún a menudo ambigua.

Programas: El significado de un programa de computadora es no ambiguo y literal, y puede ser entendido enteramente por análisis de los tokens y estructura.

Los lenguajes formales son más densos que los lenguajes naturales, así que toma más tiempo leerlos. Además, la estructura es importante, así que no siempre es mejor leer de arriba hacia abajo, de izquierda a derecha. En su lugar, aprende a analizar el programa en tu cabeza, identificando los tokens e interpretando la estructura. Finalmente, los detalles importan. Pequeños errores en ortografía y puntuación, con los cuales te puedes salir con la tuya en lenguajes naturales, pueden hacer una gran diferencia en un lenguaje formal.

1.7. Depuración

Los programadores cometen errores. Por razones caprichosas, los errores de programación se llaman *bugs* (errores) y el proceso de rastrearlos se llama *depuración* (debugging).

La programación, y especialmente la depuración, a veces saca emociones fuertes. Si estás luchando con un error difícil, podrías sentirte enojado, desalentado o avergonzado.

Hay evidencia de que las personas naturalmente responden a las computadoras como si fueran personas. Cuando funcionan bien, pensamos en ellas como compañeros de equipo, y cuando son obstinadas o groseras, respondemos a ellas de la misma manera que respondemos a personas groseras y obstinadas (Reeves y Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

Prepararse para estas reacciones podría ayudarte a lidiar con ellas. Un enfoque es pensar en la computadora como un empleado con ciertas fortalezas, como velocidad y precisión, y debilidades particulares, como falta de empatía e incapacidad para captar el panorama general.

Tu trabajo es ser un buen gerente: encontrar maneras de aprovechar las fortalezas y mitigar las debilidades. Y encontrar maneras de usar tus emociones para involucrarte con el problema, sin dejar que tus reacciones interfieran con tu capacidad de trabajar efectivamente.

Aprender a depurar puede ser frustrante, pero es una habilidad valiosa que es útil para muchas actividades más allá de la programación. Al final de cada capítulo hay una sección, como esta, con mis sugerencias para depuración. ¡Espero que ayuden!

1.8. Glosario

resolución de problemas: El proceso de formular un problema, encontrar una solución y expresarla.

lenguaje de alto nivel: Un lenguaje de programación como Python que está diseñado para ser fácil de leer y escribir para los humanos.

lenguaje de bajo nivel: Un lenguaje de programación que está diseñado para ser fácil de ejecutar para una computadora; también llamado “lenguaje de máquina” o

"lenguaje ensamblador".

portabilidad: Una propiedad de un programa que puede ejecutarse en más de un tipo de computadora.

intérprete: Un programa que lee otro programa y lo ejecuta.

prompt: Caracteres mostrados por el intérprete para indicar que está listo para recibir entrada del usuario.

programa: Un conjunto de instrucciones que especifica una computación.

declaración print: Una instrucción que hace que el intérprete de Python muestre un valor en la pantalla.

operador: Un símbolo especial que representa una computación simple como suma, multiplicación o concatenación de cadenas.

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

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

entero: Un tipo que representa números enteros.

punto flotante: Un tipo que representa números con partes fraccionarias.

cadena: Un tipo que representa secuencias de caracteres.

lenguaje natural: Cualquiera de los lenguajes que hablan las personas que evolucionaron naturalmente.

lenguaje formal: Cualquiera de los lenguajes que las personas han diseñado para propósitos específicos, como representar ideas matemáticas o programas de computadora; todos los lenguajes de programación son lenguajes formales.

token: Uno de los elementos básicos de la estructura sintáctica de un programa, análogo a una palabra en un lenguaje natural.

sintaxis: Las reglas que gobiernan la estructura de un programa.

analizar: Examinar un programa y analizar la estructura sintáctica.

bug: Un error en un programa.

depuración: El proceso de encontrar y corregir bugs.

1.9. Ejercicios

Ejercicio 1.1. Es una buena idea leer este libro frente a una computadora para que puedas probar los ejemplos mientras avanzas. Siempre que estés experimentando con una nueva característica, deberías tratar de cometer errores. Por ejemplo, en el programa "¡Hola, mundo!", ¿qué pasa si omites una de las comillas? ¿Qué pasa si omites ambas? ¿Qué pasa si escribes `print` mal? Este tipo de experimento te ayuda a recordar lo que lees; también te ayuda cuando estás programando, porque llegas a saber qué significan los mensajes de error. Es mejor cometer errores ahora y a propósito que después y accidentalmente.

1. En una declaración `print`, ¿qué pasa si omites uno de los paréntesis, o ambos?
2. Si estás tratando de imprimir una cadena, ¿qué pasa si omites una de las comillas, o ambas?
3. Puedes usar un signo menos para hacer un número negativo como `-2`. ¿Qué pasa si pones un signo más antes de un número? ¿Qué tal `2++2`?
4. En notación matemática, los ceros iniciales están bien, como en `09`. ¿Qué pasa si intentas esto en Python? ¿Qué tal `011`?
5. ¿Qué pasa si tienes dos valores sin operador entre ellos?

Ejercicio 1.2. Inicia el intérprete de Python y úsalo como calculadora.

1. ¿Cuántos segundos hay en 42 minutos 42 segundos?
2. ¿Cuántas millas hay en 10 kilómetros? Pista: hay 1.61 kilómetros en una milla.
3. Si corres una carrera de 10 kilómetros en 42 minutos 42 segundos, ¿cuál es tu ritmo promedio (tiempo por milla en minutos y segundos)? ¿Cuál es tu velocidad promedio en millas por hora?

Capítulo 2

Variables, expresiones y sentencias

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

2.1. Declaraciones de asignación

Una declaración de asignación crea una nueva variable y le da un valor:

```
>>> mensaje = 'Y ahora algo completamente diferente'
>>> n = 17
>>> pi = 3.1415926535897932
```

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

Una manera común de representar variables en papel es escribir el nombre con una flecha apuntando a su valor. Este tipo de figura se llama un *diagrama de estado* porque muestra en qué estado está cada una de las variables (piénsalo como el estado mental de la variable). La Figura 2.1 muestra el resultado del ejemplo anterior.

2.2. Nombre de Variables

Los programadores generalmente eligen nombres para sus variables que sean significativos—documentan para qué se usa la variable.

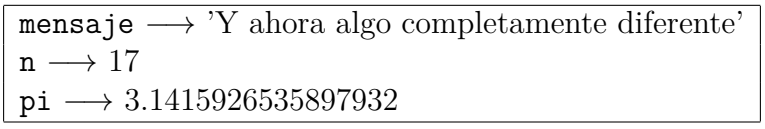

Figura 2.1: Diagrama de estado.

Los nombres de variables pueden ser tan largos como quieras. Pueden contener tanto letras como números, pero no pueden comenzar con un número. Es legal usar letras mayúsculas, pero es convencional usar solo minúsculas para nombres de variables.

El carácter guión bajo, `_`, puede aparecer en un nombre. A menudo se usa en nombres con múltiples palabras, como `tu_nombre` o `velocidad_aire_de_golondrina_sin_carga`.

Si le das a una variable un nombre ilegal, obtienes un error de sintaxis:

```
>>> 76trombones = 'gran desfile'
SyntaxError: invalid syntax
>>> mas@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Zimurgia Teórica Avanzada'
SyntaxError: invalid syntax
```

`76trombones` es ilegal porque comienza con un número. `mas@` es ilegal porque contiene un carácter ilegal, `@`. ¿Pero qué está mal con `class`?

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

Python 3 tiene estas palabras clave:

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

No tienes que memorizar esta lista. En la mayoría de entornos de desarrollo, las palabras clave se muestran en un color diferente; si tratas de usar una como nombre de variable, lo sabrás.

2.3. Expresiones y declaraciones

Una **expresión** es una combinación de valores, variables y operadores. Un valor por sí mismo se considera una expresión, y también lo es una variable, por lo que todas las siguientes son expresiones válidas:

```
>>> 42
42
>>> n
17
```

```
>>> n + 25
42
```

Cuando escribes una expresión en el prompt, el intérprete la **evalúa**, lo que significa que encuentra el valor de la expresión. En este ejemplo, `n` tiene el valor 17 y `n + 25` tiene el valor 42.

Una **declaración** (o sentencia) es una unidad de código que tiene un efecto, como crear una variable o mostrar un valor.

```
>>> n = 17
>>> print(n)
```

La primera línea es una declaración de asignación que le da un valor a `n`. La segunda línea es una declaración `print` que muestra el valor de `n`.

Cuando escribes una declaración, el intérprete la **ejecuta**, lo que significa que hace lo que sea que la declaración indique. En general, las declaraciones no tienen valores.

2.4. Modo script

Hasta ahora hemos ejecutado Python en **modo interactivo**, lo que significa que interactúas directamente con el intérprete. El modo interactivo es una buena manera de comenzar, pero si estás trabajando con más de unas pocas líneas de código, puede resultar incómodo.

La alternativa es guardar el código en un archivo llamado **script** y luego ejecutar el intérprete en **modo script** para ejecutar el script. Por convención, los scripts de Python tienen nombres que terminan con `.py`.

Si sabes cómo crear y ejecutar un script en tu computadora, estás listo para continuar. De lo contrario, recomiendo usar PythonAnywhere nuevamente. He publicado instrucciones para ejecutar en modo script en `http://tinyurl.com/thinkpython2e`.

Debido a que Python proporciona ambos modos, puedes probar fragmentos de código en modo interactivo antes de colocarlos en un script. Pero hay diferencias entre el modo interactivo y el modo script que pueden ser confusas.

Por ejemplo, si estás usando Python como una calculadora, podrías escribir:

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

La primera línea asigna un valor a `miles`, pero no tiene efecto visible. La segunda línea es una expresión, por lo que el intérprete la evalúa y muestra el resultado. Resulta que un maratón son aproximadamente 42 kilómetros.

Pero si escribes el mismo código en un script y lo ejecutas, no obtienes ninguna salida en absoluto. En modo script, una expresión por sí sola no tiene efecto visible. Python evalúa la expresión, pero no muestra el resultado. Para mostrar el resultado, necesitas una declaración `print` como esta:

```
miles = 26.2
print(miles * 1.61)
```

Este comportamiento puede ser confuso al principio.

Para verificar tu comprensión, escribe las siguientes declaraciones en el intérprete de Python y observa lo que hacen:

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

Ahora coloca las mismas declaraciones en un script y ejecútalo. ¿Cuál es la salida? Modifica el script transformando cada expresión en una declaración `print` y luego ejecútalo nuevamente.

2.5. Orden de las operaciones

Cuando una expresión contiene más de un operador, el orden de evaluación depende del **orden de las operaciones**. Para los operadores matemáticos, Python sigue la convención matemática. El acrónimo PEMDAS es una forma útil de recordar las reglas:

- **Paréntesis (Parentheses)** tienen la mayor precedencia y pueden usarse para forzar que una expresión se evalúe en el orden que desees. Dado que las expresiones entre paréntesis se evalúan primero, `2 * (3-1)` es 4, y `(1+1)**(5-2)` es 8. También puedes usar paréntesis para hacer que una expresión sea más fácil de leer, como en `(minute * 100) / 60`, incluso si no cambia el resultado.

- **Exponenciación (Exponentiation)** tiene la siguiente mayor precedencia, por lo que `1 + 2**3` es 9, no 27, y `2 * 3**2` es 18, no 36.

- **Multiplicación y División (Multiplication and Division)** tienen mayor precedencia que la Suma y la Resta. Por lo tanto, `2*3-1` es 5, no 4, y `6+4/2` es 8, no 5.

- Los **operadores con la misma precedencia** se evalúan de izquierda a derecha (excepto la exponenciación). Por lo tanto, en la expresión `degrees / 2 * pi`, la división ocurre primero y el resultado se multiplica por pi. Para dividir por 2π, puedes usar paréntesis o escribir `degrees / 2 / pi`.

No me esfuerzo mucho en recordar la precedencia de los operadores. Si no puedo determinarla solo mirando la expresión, uso paréntesis para hacerla obvia.

2.6. Operaciones con cadenas

En general, no puedes realizar operaciones matemáticas con cadenas, incluso si las cadenas parecen números, por lo que las siguientes son ilegales:

```
'chinese'-'food'
'eggs'/'easy'
'third'*'a charm'
```

Pero hay dos excepciones: `+` y `*`.

El operador `+` realiza **concatenación de cadenas**, lo que significa que une las cadenas conectándolas de extremo a extremo. Por ejemplo:

```
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
throatwarbler
```

El operador `*` también funciona con cadenas; realiza **repetición**. Por ejemplo, `'Spam'*3` es `'SpamSpamSpam'`. Si uno de los valores es una cadena, el otro tiene que ser un entero.

Este uso de `+` y `*` tiene sentido por analogía con la suma y la multiplicación. Así como `4*3` es equivalente a `4+4+4`, esperamos que `'Spam'*3` sea lo mismo que `'Spam'+'Spam'+'Spam'`, y así es.

Por otro lado, hay una diferencia significativa entre la concatenación y repetición de cadenas y la suma y multiplicación de enteros. ¿Puedes pensar en una propiedad que tiene la suma que la concatenación de cadenas no tiene?

2.7. Comentarios

A medida que los programas se vuelven más grandes y complicados, se vuelven más difíciles de leer. Los lenguajes formales son densos, y a menudo es difícil mirar un fragmento de código y determinar qué está haciendo, o por qué.

Por esta razón, es una buena idea agregar notas a tus programas para explicar en lenguaje natural lo que el programa está haciendo. Estas notas se llaman **comentarios**, y comienzan con el símbolo `#`:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

En este caso, el comentario aparece en una línea por sí mismo. También puedes poner comentarios al final de una línea:

```
percentage = (minute * 100) / 60 # percentage of an hour
```

Todo desde el # hasta el final de la línea es ignorado—no tiene efecto en la ejecución del programa.

Los comentarios son más útiles cuando documentan características no obvias del código. Es razonable asumir que el lector puede determinar qué hace el código; es más útil explicar por qué.

Este comentario es redundante con el código e inútil:

```
v = 5 # assign 5 to v
```

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

```
v = 5 # velocity in meters/second
```

Los buenos nombres de variables pueden reducir la necesidad de comentarios, pero los nombres largos pueden hacer que las expresiones complejas sean difíciles de leer, por lo que hay un equilibrio que considerar.

2.8. Depuración

En un programa pueden ocurrir tres tipos de errores: errores de sintaxis, errores de ejecución y errores semánticos. Es útil distinguir entre ellos para rastrearlos más rápidamente.

Error de sintaxis: "Sintaxis" se refiere a la estructura de un programa y las reglas sobre esa estructura. Por ejemplo, los paréntesis tienen que venir en pares coincidentes, por lo que `(1 + 2)` es legal, pero `8)` es un error de sintaxis.

Si hay un error de sintaxis en cualquier parte de tu programa, Python muestra un mensaje de error y se cierra, y no podrás ejecutar el programa. Durante las primeras semanas de tu carrera como programador, podrías pasar mucho tiempo rastreando errores de sintaxis. A medida que ganes experiencia, cometerás menos errores y los encontrarás más rápido.

Error de ejecución: El segundo tipo de error es un error de ejecución, llamado así porque el error no aparece hasta después de que el programa haya comenzado a ejecutarse. Estos errores también se llaman excepciones porque usualmente indican que algo excepcional (y malo) ha ocurrido.

Los errores de ejecución son raros en los programas simples que verás en los primeros capítulos, por lo que podría pasar un tiempo antes de que encuentres uno.

Error semántico: El tercer tipo de error es "semántico", lo que significa relacionado con el significado. Si hay un error semántico en tu programa, se ejecutará sin generar

mensajes de error, pero no hará lo correcto. Hará algo diferente. Específicamente, hará lo que le dijiste que hiciera.

Identificar errores semánticos puede ser complicado porque requiere que trabajes hacia atrás, mirando la salida del programa y tratando de averiguar qué está haciendo.

2.9. Glosario

variable: Un nombre que se refiere a un valor.

asignación: Una declaración que asigna un valor a una variable.

diagrama de estado: Una representación gráfica de un conjunto de variables y los valores a los que se refieren.

palabra clave: Una palabra reservada que se usa para analizar un programa; no puedes usar palabras clave como `if`, `def` y `while` como nombres de variables.

operando: Uno de los valores sobre los que opera un operador.

expresión: Una combinación de variables, operadores y valores que representa un resultado único.

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

declaración: Una sección de código que representa un comando o acción. Hasta ahora, las declaraciones que hemos visto son asignaciones y declaraciones `print`.

ejecutar: Ejecutar una declaración y hacer lo que dice.

modo interactivo: Una forma de usar el intérprete de Python escribiendo código en el prompt.

modo script: Una forma de usar el intérprete de Python para leer código de un script y ejecutarlo.

script: Un programa almacenado en un archivo.

orden de las operaciones: Reglas que gobiernan el orden en que se evalúan las expresiones que involucran múltiples operadores y operandos.

concatenar: Unir dos operandos de extremo a extremo.

comentario: Información en un programa que está destinada a otros programadores (o cualquiera que lea el código fuente) y no tiene efecto en la ejecución del programa.

error de sintaxis: Un error en un programa que hace imposible analizarlo (y por lo tanto imposible de interpretar).

excepción: Un error que se detecta mientras el programa se está ejecutando.

semántica: El significado de un programa.

error semántico: Un error en un programa que hace que haga algo diferente de lo que el programador pretendía.

2.10. Ejercicios

Ejercicio 2.1. Repitiendo mi consejo del capítulo anterior, siempre que aprendas una nueva característica, deberías probarla en modo interactivo y cometer errores a propósito para ver qué sale mal.

- Hemos visto que `n = 42` es legal. ¿Qué tal `42 = n`?
- ¿Qué tal `x = y = 1`?
- En algunos lenguajes, cada declaración termina con un punto y coma, `;`. ¿Qué pasa si pones un punto y coma al final de una declaración de Python?
- ¿Qué pasa si pones un punto al final de una declaración?
- En notación matemática puedes multiplicar x e y así: xy. ¿Qué pasa si intentas eso en Python?

Ejercicio 2.2. Practica usando el intérprete de Python como una calculadora:

1. El volumen de una esfera con radio r es $\frac{4}{3}\pi r^3$. ¿Cuál es el volumen de una esfera con radio 5?
2. Supón que el precio de portada de un libro es \$24.95, pero las librerías obtienen un 40 % de descuento. Los costos de envío son \$3 para la primera copia y 75 centavos para cada copia adicional. ¿Cuál es el costo total al por mayor para 60 copias?
3. Si dejo mi casa a las 6:52 am y corro 1 milla a un ritmo fácil (8:15 por milla), luego 3 millas a ritmo tempo (7:12 por milla) y 1 milla a ritmo fácil nuevamente, ¿a qué hora llego a casa para el desayuno?

Capítulo 3

Funciones

En el contexto de la programación, una **función** es una secuencia nombrada de declaraciones que realiza una computación. Cuando defines una función, especificas el nombre y la secuencia de declaraciones. Más tarde, puedes "llamar" a la función por su nombre.

3.1. Llamadas a funciones

Ya hemos visto un ejemplo de una llamada a función:

```
>>> type(42)
<class 'int'>
```

El nombre de la función es `type`. La expresión entre paréntesis se llama el **argumento** de la función. El resultado, para esta función, es el tipo del argumento.

Es común decir que una función "toma" un argumento y "devuelve" un resultado. El resultado también se llama el **valor de retorno**.

Python proporciona funciones que convierten valores de un tipo a otro. La función `int` toma cualquier valor y lo convierte a un entero, si puede, o se queja de lo contrario:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

`int` puede convertir valores de punto flotante a enteros, pero no redondea; corta la parte fraccionaria:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` convierte enteros y cadenas a números de punto flotante:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finalmente, `str` convierte su argumento a una cadena de texto:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.2. Funciones Matemáticas

Python tiene un módulo `math` que proporciona la mayoría de las funciones matemáticas familiares. Un módulo es un archivo que contiene una colección de funciones relacionadas.

Antes de poder usar las funciones en un módulo, tenemos que importarlo con una declaración de importación:

```
>>> import math
```

Esta declaración crea un objeto módulo llamado `math`. Si muestras el objeto módulo, obtienes información sobre él:

```
>>> math
<module 'math' (built-in)>
```

El objeto módulo contiene las funciones y variables definidas en el módulo. Para acceder a una de las funciones, tienes que especificar el nombre del módulo y el nombre de la función, separados por un punto (también conocido como período). Este formato se llama notación de punto.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

El primer ejemplo usa `math.log10` para calcular una relación señal-ruido en decibelios (asumiendo que `signal_power` y `noise_power` están definidos). El módulo `math` también proporciona `log`, que calcula logaritmos en base e.

El segundo ejemplo encuentra el seno de `radians`. El nombre de la variable `radians` es una pista de que `sin` y las otras funciones trigonométricas (`cos`, `tan`, etc.) toman argumentos en radianes.

Para convertir de grados a radianes, divide por 180 y multiplica por π:

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

La expresión `math.pi` obtiene la variable π del módulo `math`. Su valor es una aproximación en punto flotante de π, precisa hasta aproximadamente 15 dígitos.

Si conoces trigonometría, puedes verificar el resultado anterior comparándolo con la raíz cuadrada de dos, dividida por dos:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

3.3. Composición

Hasta ahora, hemos examinado los elementos de un programa—variables, expresiones y declaraciones—de forma aislada, sin hablar sobre cómo combinarlos.

Una de las características más útiles de los lenguajes de programación es su capacidad para tomar pequeños bloques de construcción y componerlos. Por ejemplo, el argumento de una función puede ser cualquier tipo de expresión, incluyendo operadores aritméticos:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

E incluso llamadas a funciones:

```
x = math.exp(math.log(x+1))
```

Casi en cualquier lugar donde puedas poner un valor, puedes poner una expresión arbitraria, con una excepción: el lado izquierdo de una declaración de asignación tiene que ser un nombre de variable. Cualquier otra expresión en el lado izquierdo es un error de sintaxis (veremos excepciones a esta regla más adelante).

```
>>> minutes = hours * 60
>>> hours * 60 = minutes
SyntaxError: can't assign to operator
```

3.4. Agregando nuevas funciones

Hasta ahora, solo hemos estado usando las funciones que vienen con Python, pero también es posible añadir nuevas funciones. Una definición de función especifica el nombre de una nueva función y la secuencia de declaraciones que se ejecutan cuando la función es llamada.

Aquí hay un ejemplo:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

`def` es una palabra clave que indica que esta es una definición de función. El nombre de la función es `print_lyrics`. Las reglas para los nombres de funciones son las mismas que para los nombres de variables: letras, números y guiones bajos son legales, pero el primer carácter no puede ser un número. No puedes usar una palabra clave como nombre de una función, y debes evitar tener una variable y una función con el mismo nombre.

Los paréntesis vacíos después del nombre indican que esta función no toma ningún argumento.

La primera línea de la definición de función se llama el **encabezado**; el resto se llama el **cuerpo**. El encabezado tiene que terminar con dos puntos y el cuerpo tiene que estar indentado. Por convención, la indentación es siempre de cuatro espacios. El cuerpo puede contener cualquier número de declaraciones.

Las cadenas en las declaraciones `print` están encerradas en comillas dobles. Las comillas simples y dobles hacen lo mismo; la mayoría de las personas usan comillas simples excepto en casos como este donde una comilla simple (que también es un apóstrofe) aparece en la cadena.

Todas las comillas (simples y dobles) deben ser "comillas rectas", usualmente ubicadas junto a Enter en el teclado. Las "comillas curvas", como las de esta oración, no son legales en Python.

Si escribes una definición de función en modo interactivo, el intérprete imprime puntos (...) para hacerte saber que la definición no está completa:

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...
...     print("I sleep all night and I work all day.")
```

Para terminar la función, tienes que ingresar una línea vacía.

Definir una función crea un objeto función, que tiene tipo `function`:

```
>>> print(print_lyrics)
<function print_lyrics at 0x7f...>
```

```
>>> type(print_lyrics)
<class 'function'>
```

La sintaxis para llamar a la nueva función es la misma que para las funciones incorporadas:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Una vez que has definido una función, puedes usarla dentro de otra función. Por ejemplo, para repetir el estribillo anterior, podríamos escribir una función llamada `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

Y luego llamar a `repeat_lyrics`:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Pero esa no es realmente como va la canción.

3.5. Definiciones y Usos

Reuniendo los fragmentos de código de la sección anterior, todo el programa se ve así:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

Este programa contiene dos definiciones de función: `print_lyrics` y `repeat_lyrics`. Las definiciones de función se ejecutan igual que otras declaraciones, pero el efecto es crear objetos función. Las declaraciones dentro de la función no se ejecutan hasta que la función es llamada, y la definición de función no genera ninguna salida.

Como podrías esperar, tienes que crear una función antes de poder ejecutarla. En otras palabras, la definición de función tiene que ejecutarse antes de que la función sea llamada.

Como ejercicio, mueve la última línea de este programa al principio, para que la llamada a la función aparezca antes de las definiciones. Ejecuta el programa y ve qué mensaje de error obtienes.

Ahora mueve la llamada a la función de vuelta al final y mueve la definición de `print_lyrics` después de la definición de `repeat_lyrics`. ¿Qué sucede cuando ejecutas este programa?

3.6. Flujo de Ejecución

Para asegurar que una función esté definida antes de su primer uso, tienes que conocer el orden en que se ejecutan las declaraciones, lo cual se llama el flujo de ejecución.

La ejecución siempre comienza en la primera declaración del programa. Las declaraciones se ejecutan una a la vez, en orden de arriba hacia abajo.

Las definiciones de función no alteran el flujo de ejecución del programa, pero recuerda que las declaraciones dentro de la función no se ejecutan hasta que la función es llamada.

Una llamada a función es como un desvío en el flujo de ejecución. En lugar de ir a la siguiente declaración, el flujo salta al cuerpo de la función, ejecuta las declaraciones ahí, y luego regresa para continuar donde se quedó.

Eso suena bastante simple, hasta que recuerdas que una función puede llamar a otra. Mientras está en el medio de una función, el programa podría tener que ejecutar las declaraciones en otra función. Luego, mientras ejecuta esa nueva función, ¡el programa podría tener que ejecutar aún otra función!

Afortunadamente, Python es bueno manteniendo el registro de dónde está, así que cada vez que una función se completa, el programa continúa donde se quedó en la función que la llamó. Cuando llega al final del programa, termina.

En resumen, cuando lees un programa, no siempre quieres leer de arriba hacia abajo. A veces tiene más sentido si sigues el flujo de ejecución.

3.7. Parámetros y Argumentos

Algunas de las funciones que hemos visto requieren argumentos. Por ejemplo, cuando llamas a `math.sin` pasas un número como argumento. Algunas funciones toman más de un argumento: `math.pow` toma dos, la base y el exponente.

Dentro de la función, los argumentos son asignados a variables llamadas parámetros. Aquí hay una definición para una función que toma un argumento:

```
def print_twice(bruce):
```

```
    print(bruce)
    print(bruce)
```

Esta función asigna el argumento a un parámetro llamado `bruce`. Cuando la función es llamada, imprime el valor del parámetro (sea lo que sea) dos veces.

Esta función funciona con cualquier valor que pueda ser impreso.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(42)
42
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

Las mismas reglas de composición que se aplican a las funciones incorporadas también se aplican a las funciones definidas por el programador, así que podemos usar cualquier tipo de expresión como argumento para `print_twice`:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

El argumento es evaluado antes de que la función sea llamada, así que en los ejemplos las expresiones `'Spam '*4` y `math.cos(math.pi)` son evaluadas solo una vez.

También puedes usar una variable como argumento:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

El nombre de la variable que pasamos como argumento (`michael`) no tiene nada que ver con el nombre del parámetro (`bruce`). No importa cómo se llamaba el valor en casa (en el que llama); aquí en `print_twice`, llamamos a todos `bruce`.

3.8. Variables y Parámetros Locales

Cuando creas una variable dentro de una función, es local, lo que significa que solo existe dentro de la función. Por ejemplo:

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

Esta función toma dos argumentos, los concatena, e imprime el resultado dos veces. Aquí hay un ejemplo que la usa:

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

Cuando `cat_twice` termina, la variable `cat` es destruida. Si tratamos de imprimirla, obtenemos una excepción:

___main___	
line1	⟶ 'Bing tiddle '
line2	⟶ 'tiddle bang.'

cat_twice	
part1	⟶ 'Bing tiddle '
part2	⟶ 'tiddle bang.'
cat	⟶ 'Bing tiddle tiddle bang.'

print_twice	
bruce	⟶ 'Bing tiddle tiddle bang.'

Figura 3.1: Diagrama de pila.

```
>>> print(cat)
NameError: name 'cat' is not defined
```

3.9. Diagramas de Pila

Para hacer un seguimiento de qué variables pueden ser usadas en dónde, a veces es útil dibujar un diagrama de pila. Como los diagramas de estado, los diagramas de pila muestran el valor de cada variable, pero también muestran la función a la que pertenece cada variable.

Cada función está representada por un marco. Un marco es una caja con el nombre de una función al lado y los parámetros y variables de la función dentro de ella. El diagrama de pila para el ejemplo anterior se muestra en la Figura 3.1.

Los marcos están organizados en una pila que indica qué función llamó a cuál, y así sucesivamente. En este ejemplo, `print_twice` fue llamada por `cat_twice`, y `cat_twice` fue llamada por `__main__`, que es un nombre especial para el marco superior. Cuando creas una variable fuera de cualquier función, pertenece a `__main__`.

Cada parámetro se refiere al mismo valor que su argumento correspondiente. Así, `part1` tiene el mismo valor que `line1`, `part2` tiene el mismo valor que `line2`, y `bruce` tiene el mismo valor que `cat`.

Si ocurre un error durante una llamada a función, Python imprime el nombre de la función, el nombre de la función que la llamó, y el nombre de la función que llamó a esa, todo el camino de vuelta hasta `__main__`.

Por ejemplo, si tratas de acceder a `cat` desde dentro de `print_twice`, obtienes un `NameError`:

```
Traceback (innermost last):
File "test.py", line 13, in __main__
    cat_twice(line1, line2)
File "test.py", line 5, in cat_twice
    print_twice(cat)
File "test.py", line 9, in print_twice
    print(cat)
NameError: name 'cat' is not defined
```

Esta lista de funciones se llama un *traceback*. Te dice en qué archivo de programa ocurrió el error, y en qué línea, y qué funciones se estaban ejecutando en ese momento. También muestra la línea de código que causó el error.

El orden de las funciones en el traceback es el mismo que el orden de los marcos en el diagrama de pila. La función que se está ejecutando actualmente está en la parte inferior.

3.10. Funciones con valor de retorno y funciones vacías

Algunas de las funciones que hemos usado, como las funciones matemáticas, devuelven resultados; por falta de un nombre mejor, las llamo funciones con valor de retorno. Otras funciones, como `print_twice`, realizan una acción pero no devuelven un valor. Se llaman funciones vacías.

Cuando llamas a una función con valor de retorno, casi siempre quieres hacer algo con el resultado; por ejemplo, podrías asignarlo a una variable o usarlo como parte de una

expresión:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

Cuando llamas a una función en modo interactivo, Python muestra el resultado:

```
>>> math.sqrt(5)
2.2360679774997898
```

Pero en un script, si llamas a una función con valor de retorno por sí sola, ¡el valor de retorno se pierde para siempre!

```
math.sqrt(5)
```

Este script calcula la raíz cuadrada de 5, pero como no almacena ni muestra el resultado, no es muy útil.

Las funciones vacías pueden mostrar algo en la pantalla o tener algún otro efecto, pero no tienen un valor de retorno. Si asignas el resultado a una variable, obtienes un valor especial llamado `None`.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

El valor `None` no es lo mismo que la cadena `'None'`. Es un valor especial que tiene su propio tipo:

```
>>> type(None)
<class 'NoneType'>
```

Las funciones que hemos escrito hasta ahora son todas vacías. Comenzaremos a escribir funciones con valor de retorno en unos pocos capítulos.

3.11. ¿Por qué funciones?

Puede que no esté claro por qué vale la pena tomarse la molestia de dividir un programa en funciones. Hay varias razones:

- Crear una nueva función te da la oportunidad de nombrar un grupo de instrucciones, lo que hace que tu programa sea más fácil de leer y depurar.
- Las funciones pueden hacer que un programa sea más pequeño al eliminar código repetitivo. Más tarde, si haces un cambio, solo tienes que hacerlo en un lugar.

- Dividir un programa largo en funciones te permite depurar las partes una a la vez y luego ensamblarlas en un todo funcional.
- Las funciones bien diseñadas son a menudo útiles para muchos programas. Una vez que escribes y depuras una, puedes reutilizarla.

3.12. Depuración

Una de las habilidades más importantes que adquirirás es la depuración. Aunque puede ser frustrante, la depuración es una de las partes más intelectualmente ricas, desafiantes e interesantes de la programación.

En cierto modo, la depuración es como trabajo de detective. Te enfrentas a pistas y tienes que inferir los procesos y eventos que llevaron a los resultados que ves.

La depuración también es como una ciencia experimental. Una vez que tienes una idea sobre qué está saliendo mal, modificas tu programa e intentas de nuevo. Si tu hipótesis era correcta, puedes predecir el resultado de la modificación, y das un paso más cerca de un programa que funcione. Si tu hipótesis era incorrecta, tienes que crear una nueva.

Como señaló Sherlock Holmes: Çuando has eliminado lo imposible, lo que queda, por improbable que sea, debe ser la verdad."(A. Conan Doyle, *El signo de los cuatro*)

Para algunas personas, programar y depurar son la misma cosa. Es decir, programar es el proceso de depurar gradualmente un programa hasta que haga lo que quieres. La idea es que deberías empezar con un programa que funcione y hacer pequeñas modificaciones, depurándolas a medida que avanzas.

3.13. Glosario

función: Una secuencia nombrada de instrucciones que realiza alguna operación útil. Las funciones pueden o no tomar argumentos y pueden o no producir un resultado.

definición de función: Una instrucción que crea una nueva función, especificando su nombre, parámetros y las instrucciones que contiene.

objeto función: Un valor creado por una definición de función. El nombre de la función es una variable que se refiere a un objeto función.

encabezado: La primera línea de una definición de función.

cuerpo: La secuencia de instrucciones dentro de una definición de función.

parámetro: Un nombre usado dentro de una función para referirse al valor pasado como argumento.

llamada a función: Una instrucción que ejecuta una función. Consiste en el nombre de la función seguido de una lista de argumentos entre paréntesis.

argumento: Un valor proporcionado a una función cuando se llama la función. Este valor se asigna al parámetro correspondiente en la función.

variable local: Una variable definida dentro de una función. Una variable local solo puede usarse dentro de su función.

valor de retorno: El resultado de una función. Si una llamada a función se usa como una expresión, el valor de retorno es el valor de la expresión.

función con valor de retorno: Una función que devuelve un valor.

función vacía: Una función que siempre devuelve `None`.

None: Un valor especial devuelto por las funciones vacías.

módulo: Un archivo que contiene una colección de funciones relacionadas y otras definiciones.

instrucción import: Una instrucción que lee un archivo de módulo y crea un objeto módulo.

objeto módulo: Un valor creado por una instrucción import que proporciona acceso a los valores definidos en un módulo.

notación de punto: La sintaxis para llamar una función en otro módulo especificando el nombre del módulo seguido de un punto y el nombre de la función.

composición: Usar una expresión como parte de una expresión más grande, o una instrucción como parte de una instrucción más grande.

flujo de ejecución: El orden en que se ejecutan las instrucciones.

diagrama de pila: Una representación gráfica de una pila de funciones, sus variables y los valores a los que se refieren.

marco: Una caja en un diagrama de pila que representa una llamada a función. Contiene las variables locales y parámetros de la función.

traceback: Una lista de las funciones que se están ejecutando, impresa cuando ocurre una excepción.

3.14. Ejercicios

Ejercicio 3.1. Escribe una función llamada `right_justify` que tome una cadena llamada `s` como parámetro e imprima la cadena con suficientes espacios iniciales para que la última letra de la cadena esté en la columna 70 de la pantalla.

```
>>> right_justify('monty')
                                                                monty
```

Pista: Usa concatenación y repetición de cadenas. Además, Python proporciona una función incorporada llamada `len` que devuelve la longitud de una cadena, por lo que el valor de `len('monty')` es 5.

Ejercicio 3.2. Un objeto función es un valor que puedes asignar a una variable o pasar como argumento. Por ejemplo, `do_twice` es una función que toma un objeto función como argumento y lo llama dos veces:

```
def do_twice(f):
    f()
    f()
```

Aquí tienes un ejemplo que usa `do_twice` para llamar una función llamada `print_spam` dos veces.

```
def print_spam():
    print('spam')

do_twice(print_spam)
```

1. Escribe este ejemplo en un script y pruébalo.
2. Modifica `do_twice` para que tome dos argumentos, un objeto función y un valor, y llame a la función dos veces, pasando el valor como argumento.
3. Copia la definición de `print_twice` de anteriormente en este capítulo a tu script.
4. Usa la versión modificada de `do_twice` para llamar `print_twice` dos veces, pasando `'spam'` como argumento.
5. Define una nueva función llamada `do_four` que tome un objeto función y un valor y llame a la función cuatro veces, pasando el valor como parámetro. Debería haber solo dos instrucciones en el cuerpo de esta función, no cuatro.

Solución: https://thinkpython.com/code/do_four.py.

Ejercicio 3.3. Nota: Este ejercicio debe hacerse usando solo las instrucciones y otras características que hemos aprendido hasta ahora.

1. Escribe una función que dibuje una cuadrícula como la siguiente:

```
+----+----+
|    |    |
|    |    |
|    |    |
|    |    |
+----+----+
|    |    |
|    |    |
|    |    |
|    |    |
+----+----+
```

   Pista: para imprimir más de un valor en una línea, puedes imprimir una secuencia de valores separados por comas:

```
print('+', '-')
```

   Por defecto, `print` avanza a la siguiente línea, pero puedes anular ese comportamiento y poner un espacio al final, así:

```
print('+', end=' ')
print('-')
```

   La salida de estas instrucciones es `'+-'` en la misma línea. La salida de la siguiente instrucción `print` comenzaría en la siguiente línea.

2. Escribe una función que dibuje una cuadrícula similar con cuatro filas y cuatro columnas.

Solución: https://thinkpython.com/code/grid.py.

Crédito: Este ejercicio está basado en un ejercicio en Oualline, *Practical C Programming, Third Edition*, O'Reilly Media, 1997.

Capítulo 4

Caso de estudio: diseño de interfaz

Este capítulo presenta un estudio de caso que demuestra un proceso para diseñar funciones que trabajen juntas.

Introduce el módulo `turtle`, que te permite crear imágenes usando gráficos de tortuga. El módulo `turtle` está incluido en la mayoría de las instalaciones de Python, pero si estás ejecutando Python usando PythonAnywhere, no podrás ejecutar los ejemplos de turtle (al menos no podías cuando escribí esto).

Si ya has instalado Python en tu computadora, deberías poder ejecutar los ejemplos. De lo contrario, ahora es un buen momento para instalarlo. He publicado instrucciones en `http://tinyurl.com/thinkpython2e`.

Los ejemplos de código de este capítulo están disponibles en `https://thinkpython.com/code/polygon.py`.

4.1. El modelo Tortuga

4.2. Repetición simple

4.3. Encapsulación

4.4. Generalización

4.5. Diseño de interfaz