You can order print and electronic versions of *Think Python 3e* from Bookshop.org and Amazon.

Welcome

This is the Jupyter notebook for Chapter 1 of *Think Python*, 3rd edition, by Allen B. Downey.

If you are not familiar with Jupyter notebooks, click here for a short introduction.

Then, if you are not already running this notebook on Colab, click here to run this notebook on Colab.

The following cell downloads a file and runs some code that is used specifically for this book. You don't have to understand this code yet, but you should run it before you do anything else in this notebook. Remember that you can run the code by selecting the cell and pressing the play button (a triangle in a circle) or hold down `Shift` and press `Enter`.

```
from os.path import basename, exists

def download(url):
    filename = basename(url)
    if not exists(filename):
        from urllib.request import urlretrieve

        local, _ = urlretrieve(url, filename)
        print("Downloaded " + str(local))
    return filename

download('https://github.com/AllenDowney/ThinkPython/raw/v3/thinkpytho
n.py');

import thinkpython
```

```
Downloaded thinkpython.py
```

Programming as a way of thinking

The first goal of this book is to teach you how to program in Python. But learning to program means learning a new way to think, so the second goal of this book is to help you think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas -- specifically computations. Like engineers, they design things, assembling components into systems and evaluating trade-offs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

We will start with the most basic elements of programming and work our way up. In this chapter, we'll see how Python represents numbers, letters, and words. And you'll learn to perform arithmetic operations.

You will also start to learn the vocabulary of programming, including terms like operator, expression, value, and type. This vocabulary is important -- you will need it to understand the rest of the book, to communicate with other programmers, and to use and understand virtual assistants.

Arithmetic operators

An **arithmetic operator** is a symbol that represents an arithmetic computation. For example, the plus sign, `+`, performs addition.

```
30 + 12
```

The minus sign, `-`, is the operator that performs subtraction.

```
43 - 1
```

The asterisk, `*`, performs multiplication.

```
6 * 7
```

And the forward slash, `/`, performs division:

```
84 / 2
```

Notice that the result of the division is `42.0` rather than `42`. That's because there are two types of numbers in Python:

- **integers**, which represent whole numbers, and
- **floating-point numbers**, which represent numbers with a decimal point.

If you add, subtract, or multiply two integers, the result is an integer. But if you divide two integers, the result is a floating-point number. Python provides another operator, `//`, that performs **integer division**. The result of integer division is always an integer.

```
84 // 2
```

Integer division is also called "floor division" because it always rounds down (toward the "floor").

```
85 // 2
```

Finally, the operator `**` performs exponentiation; that is, it raises a number to a power:

```
7 ** 2
```

In some other languages, the caret, `^`, is used for exponentiation, but in Python it is a bitwise operator called XOR. If you are not familiar with bitwise operators, the result might be unexpected:

```
7 ^ 2
```

I won't cover bitwise operators in this book, but you can read about them at http://wiki.python.org/moin/BitwiseOperators.

Expressions

A collection of operators and numbers is called an **expression**. An expression can contain any number of operators and numbers. For example, here's an expression that contains two operators.

```
6 + 6 ** 2
```

Notice that exponentiation happens before addition. Python follows the order of operations you might have learned in a math class: exponentiation happens before multiplication and division, which happen before addition and subtraction.

In the following example, multiplication happens before addition.

```
12 + 5 * 6
```

If you want the addition to happen first, you can use parentheses.

```
(12 + 5) * 6
```

Every expression has a **value**. For example, the expression `6 * 7` has the value `42`.

Arithmetic functions

In addition to the arithmetic operators, Python provides a few **functions** that work with numbers. For example, the `round` function takes a floating-point number and rounds it off to the nearest whole number.

```
round(42.4)

round(42.6)
```

The `abs` function computes the absolute value of a number. For a positive number, the absolute value is the number itself.

```
abs(42)
```

For a negative number, the absolute value is positive.

```
abs(-42)
```

When we use a function like this, we say we're **calling** the function. An expression that calls a function is a **function call**.

When you call a function, the parentheses are required. If you leave them out, you get an error message.

NOTE: The following cell uses `%%expect`, which is a Jupyter "magic command" that means we expect the code in this cell to produce an error. For more on this topic, see the Jupyter notebook introduction.

```
%%expect SyntaxError

abs 42
```

You can ignore the first line of this message; it doesn't contain any information we need to understand right now. The second line is the code that contains the error, with a caret (`^`) beneath it to indicate where the error was discovered.

The last line indicates that this is a **syntax error**, which means that there is something wrong with the structure of the expression. In this example, the problem is that a function call requires parentheses.

Let's see what happens if you leave out the parentheses *and* the value.

```
abs

<function abs(x, /)>
```

A function name all by itself is a legal expression that has a value. When it's displayed, the value indicates that `abs` is a function, and it includes some additional information I'll explain later.

Strings

In addition to numbers, Python can also represent sequences of letters, which are called **strings** because the letters are strung together like beads on a necklace. To write a string, we can put a sequence of letters inside straight quotation marks.

```
'Hello'
```

It is also legal to use double quotation marks.

```
"world"
```

Double quotes make it easy to write a string that contains an apostrophe, which is the same symbol as a straight quote.

```
"it's a small "
```

Strings can also contain spaces, punctuation, and digits.

```
'Well, '
```

The `+` operator works with strings; it joins two strings into a single string, which is called **concatenation**

```
'Well, ' + "it's a small " + 'world.'
```

The `*` operator also works with strings; it makes multiple copies of a string and concatenates them.

```
'Spam, ' * 4
```

The other arithmetic operators don't work with strings.

Python provides a function called `len` that computes the length of a string.`

```
len('Spam')
```

Notice that `len` counts the letters between the quotes, but not the quotes.

When you create a string, be sure to use straight quotes. The back quote, also known as a backtick, causes a syntax error.

```
%%expect SyntaxError

`Hello`
```

```
  File "<ipython-input-2-45308b825ee8>", line 1
    `Hello`
    ^
SyntaxError: invalid syntax
```

Smart quotes, also known as curly quotes, are also illegal.

```
%%expect SyntaxError

‘Hello’
```

Values and types

So far we've seen three kinds of values:

- `2` is an integer,

- `42.0` is a floating-point number, and
- `'Hello'` is a string.

A kind of value is called a **type**. Every value has a type -- or we sometimes say it "belongs to" a type.

Python provides a function called `type` that tells you the type of any value. The type of an integer is `int`.

```
type(2)
```

The type of a floating-point number is `float`.

```
type(42.0)
```

And the type of a string is `str`.

```
type('Hello, World!')
```

The types `int`, `float`, and `str` can be used as functions. For example, `int` can take a floating-point number and convert it to an integer (always rounding down).

```
int(42.9)
```

And `float` can convert an integer to a floating-point value.

```
float(42)
```

Now, here's something that can be confusing. What do you get if you put a sequence of digits in quotes?

```
'126'
```

It looks like a number, but it is actually a string.

```
type('126')
```

If you try to use it like a number, you might get an error.

```
%%expect TypeError

'126' / 3
```

This example generates a `TypeError`, which means that the values in the expression, which are called **operands**, have the wrong type. The error message indicates that the `/` operator does not support the types of these values, which are `str` and `int`.

If you have a string that contains digits, you can use `int` to convert it to an integer.

```
int('126') / 3
```

If you have a string that contains digits and a decimal point, you can use `float` to convert it to a floating-point number.

```
float('12.6')
```

When you write a large integer, you might be tempted to use commas between groups of digits, as in `1,000,000`. This is a legal expression in Python, but the result is not an integer.

```
1,000,000
```

Python interprets `1,000,000` as a comma-separated sequence of integers. We'll learn more about this kind of sequence later.

You can use underscores to make large numbers easier to read.

```
1_000_000
```

Formal and natural languages

Natural languages are the languages people speak, like English, Spanish, and French. They were not designed by people; they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Similarly, programming languages are formal languages that have been designed to express computations.

Although formal and natural languages have some features in common there are important differences:

- Ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any program has exactly one meaning, regardless of context.
- Redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages use redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.
- Literalness: Natural languages are full of idiom and metaphor. Formal languages mean exactly what they say.

Because we all grow up speaking natural languages, it is sometimes hard to adjust to formal languages. Formal languages are more dense than natural languages, so it takes longer to read them. Also, the structure is important, so it is not always best to read from top to bottom, left to

right. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

Debugging

Programmers make mistakes. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down is called **debugging**.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, sad, or embarrassed.

Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a section, like this one, with my suggestions for debugging. I hope they help!

Glossary

arithmetic operator: A symbol, like `+` and `*`, that denotes an arithmetic operation like addition or multiplication.

integer: A type that represents whole numbers.

floating-point: A type that represents numbers with fractional parts.

integer division: An operator, `//`, that divides two numbers and rounds down to an integer.

expression: A combination of variables, values, and operators.

value: An integer, floating-point number, or string -- or one of other kinds of values we will see later.

function: A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

function call: An expression -- or part of an expression -- that runs a function. It consists of the function name followed by an argument list in parentheses.

syntax error: An error in a program that makes it impossible to parse -- and therefore impossible to run.

string: A type that represents sequences of characters.

concatenation: Joining two strings end-to-end.

type: A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), and strings (type `str`).

operand: One of the values on which an operator operates.

natural language: Any of the languages that people speak that evolved naturally.

formal language: Any of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs. All programming languages are formal languages.

bug: An error in a program.

debugging: The process of finding and correcting errors.

Exercises

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

```
Exception reporting mode: Verbose
```

Ask a virtual assistant

As you work through this book, there are several ways you can use a virtual assistant or chatbot to help you learn.

- If you want to learn more about a topic in the chapter, or anything is unclear, you can ask for an explanation.
- If you are having a hard time with any of the exercises, you can ask for help.

In each chapter, I'll suggest exercises you can do with a virtual assistant, but I encourage you to try things on your own and see what works for you.

Here are some topics you could ask a virtual assistant about:

- Earlier I mentioned bitwise operators but I didn't explain why the value of `7 ^ 2` is 5. Try asking "What are the bitwise operators in Python?" or "What is the value of `7 XOR 2`?"
- I also mentioned the order of operations. For more details, ask "What is the order of operations in Python?"
- The `round` function, which we used to round a floating-point number to the nearest whole number, can take a second argument. Try asking "What are the arguments of the round function?" or "How do I round pi off to three decimal places?"
- There's one more arithmetic operator I didn't mention; try asking "What is the modulus operator in Python?"

Most virtual assistants know about Python, so they answer questions like this pretty reliably. But remember that these tools make mistakes. If you get code from a chatbot, test it!

Exercise

You might wonder what `round` does if a number ends in `0.5`. The answer is that it sometimes rounds up and sometimes rounds down. Try these examples and see if you can figure out what rule it follows.

```
round(42.5)
```

```
42
```

```
round(43.5)
# It seems like round() rounds numbers ending in 0.5 to the nearest
even whole number.
```

```
44
```

If you are curious, ask a virtual assistant, "If a number ends in 0.5, does Python round up or down?"

Exercise

When you learn about a new feature, you should try it out and make mistakes on purpose. That way, you learn the error messages, and when you see them again, you will know what they mean. It is better to make mistakes now and deliberately than later and accidentally.

1. You can use a minus sign to make a negative number like `-2`. What happens if you put a plus sign before a number? What about `2++2`?
2. What happens if you have two values with no operator between them, like `4 2`?
3. If you call a function like `round(42.5)`, what happens if you leave out one or both parentheses?

```
+2 # works as positive 2
2++2 # works as 2 plus positive 2, i.e., 4
4 2 # SyntaxError
round(42.5 #SyntaxError
```

```
  Cell In[9], line 4
    round(42.5 #SyntaxError
                           ^
SyntaxError: incomplete input
```

Exercise

Recall that every expression has a value, every value has a type, and we can use the `type` function to find the type of any value.

What is the type of the value of the following expressions? Make your best guess for each one, and then use `type` to find out.

- `765`
- `2.718`
- `'2 pi'`
- `abs(-7)`
- `abs(-7.0)`
- `abs`
- `int`
- `type`

```
type(765)       # guess: int □
type(2.718)     # guess: float □
type('2 pi')    # guess: str □
type(abs(-7))   # guess: int □
type(abs(-7.0)) # guess: float □
type(abs)       # guess: function □ builtin_function_or_method
type(int)       # guess: builtin_function_or_method □ type
type(type)      # guess: builtin_function_or_method □ type

type
```

Exercise

The following questions give you a chance to practice writing arithmetic expressions.

1. How many seconds are there in 42 minutes 42 seconds?
2. How many miles are there in 10 kilometers? Hint: there are 1.61 kilometers in a mile.
3. If you run a 10 kilometer race in 42 minutes 42 seconds, what is your average pace in seconds per mile?
4. What is your average pace in minutes and seconds per mile?
5. What is your average speed in miles per hour?

If you already know about variables, you can use them for this exercise. If you don't, you can do the exercise without them -- and then we'll see them in the next chapter.

```
# Solution goes here
str(42*60 + 42) + ' seconds'
```

```
'2562 seconds'
```

```
# Solution goes here
str(10 / 1.61) + ' miles'
```

```
'6.211180124223602 miles'
```

```
str((42*60+60) / (10/1.61)) + ' seconds per mile'
```

```
415.38000000000005
```

```
# Solution goes here
time = (42*60+60) / (10/1.61)
mins = time // 60
secs = time % 60
f'{mins} minutes and {secs} seconds per mile'
```

```
'6.0 minutes and 55.38000000000005 seconds per mile'
```

```
# Solution goes here
miles = (10 / 1.61)
hours = 42/60 + 42/3600
f'{miles/hours} miles per hour'
```

```
'8.727653570337614 miles per hour'
```