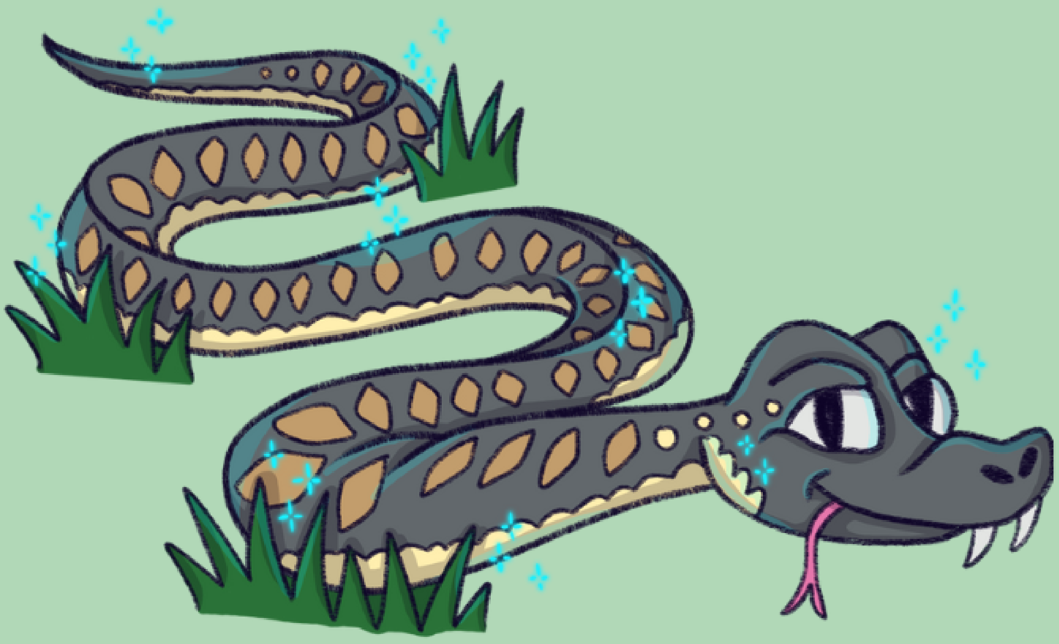
learnbyexample

100 Page Python Intro



Sundeep Agarwal

Table of contents

Preface

This book is a short, introductory guide for the Python programming language. This book is well suited:

- As a reference material for Python beginner workshops
- If you have prior experience with another programming language
- If you want a complement resource after reading a Python basics book, watching a video course, etc

Prerequisites

You should be already familiar with basic programming concepts. If you are new to programming, check out my comprehensive curated list on Python to get started.

You are also expected to get comfortable with reading manuals, searching online, visiting external links provided for further reading, tinkering with the illustrated examples, asking for help when you are stuck and so on. In other words, be proactive and curious instead of just consuming the content passively.

Conventions

- The examples presented here have been tested with **Python version 3.13.0** and includes features that are not available in earlier versions.
- Code snippets that are copy pasted from the Python REPL shell have been modified for presentation purposes. For example, comments to provide context and explanations, blank lines and shortened error messages to improve readability and so on.
- A comment with filename will be shown as the first line for program files.
- External links are provided for further exploration throughout the book. They have been chosen with care to provide more detailed resources as well as resources on related topics.
- The 100_page_python_intro repo has all the programs and files presented in this book, organized by chapter for convenience.
- Visit Exercises.md to view all the exercises from this book. To interactively practice these exercises, see my PythonExercises repo.

Acknowledgements

- Official Python website — documentation and examples
- stackoverflow and unix.stackexchange — for getting answers to pertinent questions on Python, Shell and programming in general
- /r/learnpython and /r/learnprogramming — helpful forum for beginners
- /r/Python/ — general Python discussion
- tex.stackexchange — for help on pandoc and `tex` related questions
- canva — cover image
- oxipng, pngquant and svgcleaner — optimizing images
- Warning and Info icons by Amada44 under public domain
- **Dean Clark** and **Elijah** for catching a few typos

Feedback and Errata

I would highly appreciate it if you'd let me know how you felt about this book. It could be anything from a simple thank you, pointing out a typo, mistakes in code snippets, which aspects of the book worked for you (or didn't!) and so on. Reader feedback is essential and especially so for self-published authors.

You can reach me via:

- Issue Manager: https://github.com/learnbyexample/100_page_python_intro/issues
- E-mail: learnbyexample.net@gmail.com
- Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a lazy being who prefers to work just enough to support his modest lifestyle. He accumulated vast wealth working as a Design Engineer at Analog Devices and retired from the corporate world at the ripe age of twenty-eight. Unfortunately, he squandered his savings within a few years and had to scramble trying to earn a living. Against all odds, selling programming ebooks saved his lazy self from having to look for a job again. He can now afford all the fantasy ebooks he wants to read and spends unhealthy amount of time browsing the internet.

When the creative muse strikes, he can be found working on yet another programming ebook (which invariably ends up having at least one example with regular expressions). Researching materials for his ebooks and everyday social media usage drowned his bookmarks, so he maintains curated resource lists for sanity sake. He is thankful for free learning resources and open source tools. His own contributions can be found at https://github.com/learnbyexample.

List of books: https://learnbyexample.github.io/books/

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Code snippets are available under MIT License.

Resources mentioned in the Acknowledgements section above are available under original licenses.

Book version

2.0

See Version_changes.md to track changes across book versions.

Introduction

Wikipedia does a great job of describing about Python in a few words. So, I'll just copy-paste the relevant information here:

> Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation.
> Python is dynamically type-checked and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library.
> Python consistently ranks as one of the most popular programming languages, and has gained widespread use in the machine learning community.

See also docs.python: General Python FAQ for answers to questions like "What is Python?", "What is Python good for?", "Why is it called Python?" and so on.

Installation

On modern Linux distributions, you are likely to find Python already installed. It may be a few versions behind, but should work just fine for most of the topics covered in this book. To get the exact version used here, visit the Python downloads page and install using the appropriate source for your operating system.

Using the installer from the downloads page is the easiest option to get started on Windows and macOS. See docs.python: Python Setup and Usage for more information.

For Linux, check your distribution repository first. You can also build it from source as shown below for Debian-like distributions:

```
$ wget https://www.python.org/ftp/python/3.13.0/Python-3.13.0.tar.xz
$ tar -Jxf Python-3.13.0.tar.xz
$ cd Python-3.13.0
$ ./configure --enable-optimizations
$ make
$ sudo make altinstall
```

You may have to install dependencies first, see this stackoverflow thread for details.

> See docs.python: What's New to track changes across versions.

Online tools

In case you are facing installation issues, or do not want to (or cannot) install Python on your computer for some reason, there are options to execute Python programs using online tools. Some of them are listed below:

- Repl.it — Code, collaborate, compile, run, share, and deploy Python and more online from your browser

- Pythontutor — Visualize code execution, also has example codes and ability to share sessions
- PythonAnywhere — Host, run, and code Python in the cloud

The official Python website also has a *Launch Interactive Shell* option (https://www.python.org/shell/), which gives access to a REPL session.

First program

It is customary to start learning a new programming language by printing a simple phrase. Create a new directory, say `python_programs` for this book. Then, create a plain text file named `hello.py` with your favorite text editor and type the following piece of code.

```
# hello.py
print('*************')
print('Hello there!')
print('*************')
```

If you are familiar with using the command line on a Unix-like system, run the script as shown below (use `py hello.py` if you are using Windows CMD). Other options to execute a Python program will be discussed in the next section.

```
$ python3.13 hello.py
*************
Hello there!
*************
```

A few things to note here. The first line is a comment, used here to show the name of the Python program. `print()` is a built-in function, which can be used without having to load some library. A single string argument has been used for each of the three invocations. `print()` automatically appends a newline character by default. The program ran without a compilation step. As quoted earlier, Python is an *interpreted* language. More details will be discussed in later chapters.

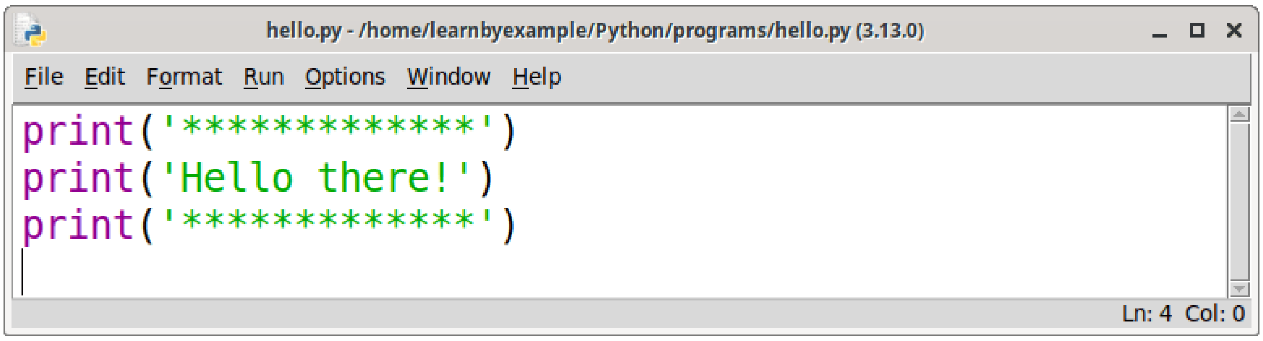
See Python behind the scenes and this list of resources if you are interested to learn inner details about Python program execution.

All the Python programs discussed in this book, along with related text files, can be accessed from my GitHub repo learnbyexample: 100_page_python_intro. However, I'd highly recommend typing the programs manually by yourself.

IDE and text editors

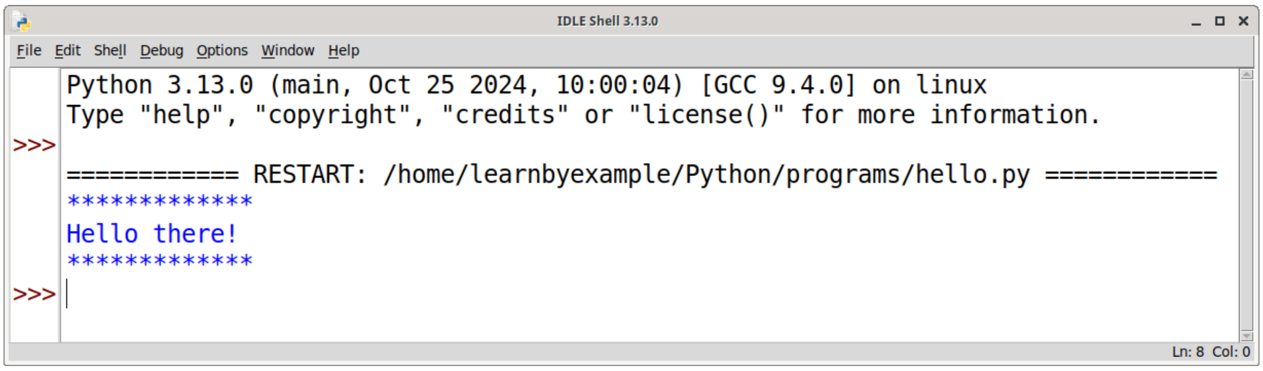
An **integrated development environment** (IDE) might suit you better if you are not comfortable with the command line. IDE provides features likes debugging, syntax highlighting, autocompletion, code refactoring and so on. They also help in setting up a **virtual environment** to manage different versions of Python and modules (more on that later). See wikipedia: IDE for more details.

If you install Python on Windows, it will automatically include **IDLE**, an IDE built using Python's `tkinter` module. On Linux, you might already have the `idle3.13` program if you installed Python manually. Otherwise you may have to install it separately.


hello.py - /home/learnbyexample/Python/programs/hello.py (3.13.0)
File Edit Format Run Options Window Help
Ln: 4 Col: 0

```
print('*************')
print('Hello there!')
print('*************')
```

When you open IDLE, you'll get a Python shell (discussed in the next section). For now, click the **New File** option under **File** menu to open a text editor. Type the short program `hello.py` discussed in the previous section. After saving the code, press **F5** to run it. You'll see the results in the shell window as shown below.



```
Python 3.13.0 (main, Oct 25 2024, 10:00:04) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============ RESTART: /home/learnbyexample/Python/programs/hello.py ============
*************
Hello there!
*************
>>>
```

Popular alternatives to IDLE are listed below:

- Thonny — Python IDE for beginners, lots of handy features like viewing variables, debugger, step through, highlight syntax errors, name completion, etc
- Pycharm — smart code completion, code inspections, on-the-fly error highlighting and quick-fixes, automated code refactorings, rich navigation capabilities, support for frameworks, etc
- Spyder — typically used for scientific computing
- Jupyter — web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text
- VSCodium — community-driven, freely-licensed binary distribution of VSCode
- Vim, Emacs, Geany, GNOME Text Editor — text editors with support for syntax highlighting and more

REPL

One of the best features of Python is the interactive shell. Such shells are also referred to as REPL, an abbreviation for **R**ead **E**valuate **P**rint **L**oop. The Python REPL makes it easy for beginners to try out code snippets for learning purposes. Beyond learning, it is also useful for developing a program in small steps, debugging a large program by trying out few lines of code at a time and so on. REPL will be used frequently in this book to show code snippets.

When you launch Python from the command line, or open IDLE, you get a shell that is ready for user input after the `>>>` prompt.

```
$ python3.13
Python 3.13.0 (main, Oct 25 2024, 10:00:04) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Try the below instructions. The first one displays a greeting using the `print()` function. Then, a user defined variable is used to store a string value. To display the value, you can either use `print()` again or just type the variable name. Expression results are immediately displayed in the shell. Name of a variable by itself is a valid expression. This behavior is unique to the REPL and an expression by itself won't display anything when used inside a script.

```
>>> print('have a nice day')
have a nice day

>>> username = 'learnbyexample'
>>> print(username)
learnbyexample

# use # to start a single line comment
# note that string representation is shown instead of actual value
# details will be discussed later
>>> username
'learnbyexample'

# use exit() to close the shell, can also use Ctrl+D shortcut
>>> exit()
```

I'll stress again the importance of following along the code snippets by manually typing them on your computer. Programming requires hands-on experience too, reading alone isn't enough. As an analogy, can you learn to drive a car by just reading about it? Since one of the prerequisite is that you should already be familiar with programming basics, I'll extend the analogy to learning to drive a different car model. Or, perhaps a different vehicle such as a truck or a bus might be more appropriate here.

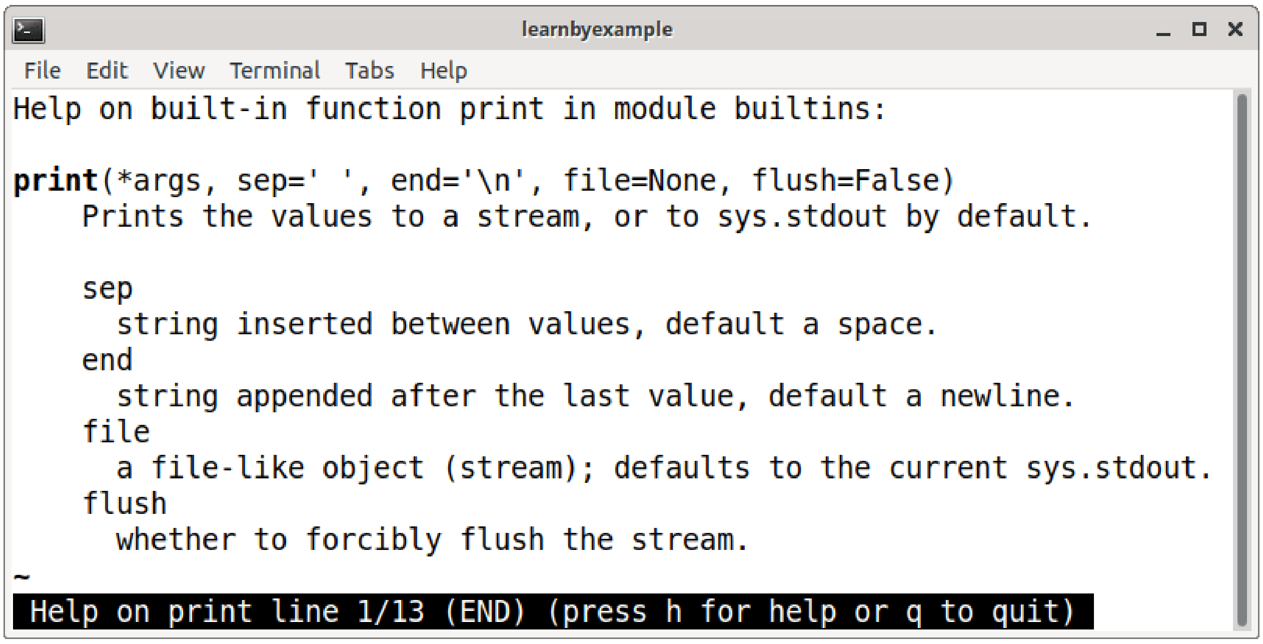
Unlike previous versions, the Python REPL now implements editing and navigation features on its own instead of relying on an external `readline` library. See REPL-acing the default REPL (PEP 762) for more information.

You can use `python3.13 -q` to avoid the *version and copyright messages* when you start an interactive shell. Use `python3.13 -h` or visit docs.python: Command line and environment for documentation on CLI options.

Documentation and getting help

The official Python website has an extensive documentation located at https://docs.python.org/3/. This includes a tutorial (which is much more comprehensive than the contents presented in this book), several guides for specific modules like `re` and `argparse` and various other information.

Python also provides a `help()` function, which is quite handy to use from the REPL. If you type `help(print)` and press the Enter key, you'll get a screen as shown below. If you are using IDLE, the output would be displayed on the same screen. Otherwise, the content might be shown on a different screen depending on your `pager` settings. Typically, pressing the `q` key will quit the `pager` and get you back to the shell.



```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
~
 Help on print line 1/13 (END) (press h for help or q to quit)
```

Quotes are necessary, for example `help('import')` and `help('del')`, if the topic you are looking for isn't an object.

If you get stuck with a problem, there are several ways to get it resolved. For example:

1. research the topic via documentation/books/tutorials/etc
2. reduce the code as much as possible so that you are left with minimal code necessary to reproduce the issue
3. talk about the problem with a friend/colleague/inanimate-objects/etc (see Rubber duck debugging)
4. search about the problem online

You can also ask for help on forums. Make sure to read the instructions provided by the respective forums before asking a question. Here are some forums you can use:

- /r/learnpython and /r/learnprogramming/ — beginner friendly
- python-forum — dedicated Python forum, encourages back and forth discussions based on the topic of the thread
- /r/Python/ — general Python discussion
- stackoverflow: python tag

The Debugging chapter will discuss more on this topic.

Numeric data types

Python is a dynamically typed language. The interpreter infers the data type of a value based on pre-determined rules. In the previous chapter, **string** values were coded using single quotes around a sequence of characters. Similarly, there are rules by which you can declare different numeric data types.

int

Integer numbers are made up of digits `0` to `9` and can be prefixed with **unary** operators like `+` or `-` . There is no restriction to the size of numbers that can be used, only limited by the memory available on your system. Here are some examples:

```
>>> 42
42
>>> 0
0
>>> +100
100
>>> -5
-5
```

For readability purposes, you can use underscores in between the digits.

```
>>> 1_000_000_000
1000000000
```

Underscore cannot be used as the first or last character, and cannot be used consecutively.

float

Here are some examples for floating-point numbers.

```
>>> 3.14
3.14
>>> -1.12
-1.12
```

Python also supports the exponential notation. See wikipedia: E scientific notation for details about this form of expressing numbers.

```
>>> 543.15e20
5.4315e+22
>>> 1.5e-5
1.5e-05
```

Unlike integers, floating-point numbers have a limited precision. While displaying very small or very large floating-point numbers, Python will automatically convert them to the exponential notation.

```
>>> 0.0000000001234567890123456789
1.2345678901234568e-10
>>> 31415926535897935809629384623048923.649234324234
3.1415926535897936e+34
```

You might also get seemingly strange results as shown below. See docs.python: Floating Point Arithmetic Issues and Limitations, stackoverflow: Is floating point math broken? and Examples of floating point problems for details and workarounds.

```
>>> 3.14 + 2
5.140000000000001
```

Arithmetic operators

All arithmetic operators you'd typically expect are available. If any operand is a floating-point number, result will be of `float` data type. Use `+` for addition, `-` for subtraction, `*` for multiplication and `**` for exponentiation. As mentioned before, REPL is quite useful for learning purposes. It makes for a good calculator for number crunching. You can also use `_` to refer to the result of the previous expression (this is applicable only in the REPL, not in Python scripts).

```
>>> 25 + 17
42
>>> 10 - 8
2
>>> 25 * 3.3
82.5
>>> 32 ** 42
1645504557321206042154969182557350504982735865633579863348609024

>>> 5 + 2
7
>>> _ * 3
21
```

There are two operators for division. Use `/` if you want a floating-point result. Using `//` between two integers will give only the integer portion of the result (no rounding).

```
>>> 4.5 / 1.5
3.0
>>> 5 / 3
1.6666666666666667
>>> 5 // 3
1
```

Use the modulo operator `%` to get the remainder. Sign of the result is same as the sign of the second operand.

```
>>> 5 % 3
2
```

```
>>> -5 % 3
1
>>> 5 % -3
-1
>>> 6.5 % -3
-2.5
```

See docs.python: Binary arithmetic operations and stackoverflow: modulo operation on negative numbers for more details.

Operator precedence

Arithmetic operator precedence follows the familiar **PEMDAS** or **BODMAS** abbreviations. Precedence, higher to lower is listed below:

- Expression inside parentheses
- exponentiation
- multiplication, division, modulo
- addition, subtraction

Expression is evaluated left-to-right when operators have the same precedence. Unary operator precedence is between exponentiation and multiplication/division operators. See docs.python: Operator precedence for complete details.

Integer formats

The integer examples so far have been coded using base 10, also known as the **decimal** format. Python has provision for representing **binary**, **octal** and **hexadecimal** formats as well. To distinguish between these different formats, a prefix is used:

- `0b` or `0B` for binary
- `0o` or `0O` for octal
- `0x` or `0X` for hexadecimal

All of these four formats fall under the `int` data type. Python displays them in decimal format by default. Underscores can be used for readability for any of these formats.

```
>>> 0b1000_1111
143
>>> 0o10
8
>>> 0x10
16

>>> 5 + 0xa
15
```

Decimal format numbers cannot be prefixed by `0` , other than `0` itself.

```
>>> 00000
0
```

```
>>> 09
  File "<python-input-1>", line 1
    09
    ^
SyntaxError: leading zeros in decimal integer literals are not permitted;
             use an 0o prefix for octal integers
```

If code execution hits a snag, you'll get an error message along with the code snippet that the interpreter thinks caused the issue. In Python parlance, an **exception** has occurred. The exception has a name (`SyntaxError` in the above example) followed by the error message. See the Exception handling chapter for more details.

Other numeric types

Python's standard data type also includes complex type (imaginary part is suffixed with the character `j`). Others like `decimal` and `fractions` are provided as modules.

- docs.python: complex
- docs.python: decimal
- docs.python: fractions

Some of the numeric types can have alphabets like `e` , `b` , `j` , etc in their values. Which implies that you cannot use variable names beginning with a number. Otherwise, it would be impossible to evaluate an expression like `result = input_value + 0x12 - 2j` .

There are many third-party libraries useful for number crunching in mathematical and engineering applications. See my list py_resources: Scientific computing for curated resources.

Strings and user input

This chapter will discuss various ways to specify string literals. After that, you'll see how to get input data from the user and handle type conversions.

Single and double quoted strings

The most common way to declare string literals is by enclosing a sequence of characters within single or double quotes. Unlike other scripting languages like **Bash**, **Perl** and **Ruby**, there is no feature difference between these forms.

REPL will again be used predominantly in this chapter. One important detail to note is that the result of an expression is displayed using the syntax of that particular data type. Use `print()` function when you want to see how a string literal looks visually.

```
>>> 'hello'
'hello'
>>> print("world")
world
```

If the string literal itself contains single or double quote characters, the other form can be used.

```
>>> print('"Will you come?" he asked.')
"Will you come?" he asked.

>>> print("it's a fine sunny day")
it's a fine sunny day
```

What to do if a string literal has both single and double quotes? You can use the `\` character to escape the quote characters. In the below examples, `\'` and `\"` will evaluate to `'` and `"` characters respectively, instead of prematurely terminating the string definition. Use `\\` if a literal backslash character is needed.

```
>>> print('"It\'s so pretty!" can I get one?')
"It's so pretty!" can I get one?

>>> print("\"It's so pretty!\" can I get one?")
"It's so pretty!" can I get one?
```

In general, the backslash character is used to construct escape sequences. For example, `\n` represents the newline character, `\t` is for the tab character and so on. You can use `\ooo` and `\xhh` to represent 256 characters in octal and hexadecimal formats respectively. For Unicode characters, you can use `\N{name}` , `\uxxxx` and `\Uxxxxxxxx` formats. See docs.python: String and Bytes literals for the full list of escape sequences and details about undefined ones.

```
>>> greeting = 'hi there.\nhow are you?'
>>> greeting
'hi there.\nhow are you?'
>>> print(greeting)
hi there.
how are you?
```

```
>>> print('item\tquantity')
item    quantity

>>> print('\u03b1\u03bb\u03b5\N{LATIN SMALL LETTER TURNED DELTA}')
αλεƍ
```

Triple quoted strings

You can also declare multiline strings by enclosing the value with three single/double quote characters. If backslash is the last character of a line, then a newline won't be inserted at that position. Here's a Python program named `triple_quotes.py` to illustrate this concept.

```
# triple_quotes.py
print('''hi there.
how are you?''')

student = '''\
Name:\tlearnbyexample
Age:\t25
Dept:\tCSE'''

print(student)
```

Here's the output of the above script:

```
$ python3.13 triple_quotes.py
hi there.
how are you?
Name:   learnbyexample
Age:    25
Dept:   CSE
```

See the Docstrings section for another use of triple quoted strings.

Raw strings

For certain cases, escape sequences would be too much of a hindrance to workaround. For example, filepaths in Windows use `\` as the delimiter. Another would be regular expressions, where the backslash character has yet another special meaning. Python provides a **raw** string syntax, where all the characters are treated literally. This form, also known as **r-strings** for short, requires a `r` or `R` character prefix to quoted strings. Forms like triple quoted strings and raw strings are for user convenience. Internally, there's just a single representation for string literals.

```
>>> print(r'item\tquantity')
item\tquantity

>>> r'item\tquantity'
'item\\tquantity'
```

```
>>> r'C:\Documents\blog\monsoon_trip.txt'
'C:\\Documents\\blog\\monsoon_trip.txt'
```

Here's an example with the `re` built-in module. The `import` statement used below will be discussed in the Importing and creating modules chapter. See my book Understanding Python re(gex)? for details on regular expressions.

```
>>> import re

# numbers >= 100 with optional leading zeros
# you'd need \\b and \\d with normal strings
>>> re.findall(r'\b0*+\d{3,}\b', '0501 035 154 12 26 98234')
['0501', '154', '98234']
```

String operators

Python provides a wide variety of features to work with strings. This chapter introduces some of them, like the `+` and `*` operators in this section. Here are some examples to concatenate strings using the `+` operator. The operands can be any expression that results in a string value and you can use any of the different ways to specify a string literal.

```
>>> str1 = 'hello'
>>> str2 = ' world'
>>> str3 = str1 + str2
>>> print(str3)
hello world

>>> str3 + r'. 1\n2'
'hello world. 1\\n2'
```

Another way to concatenate is to simply place string literals next to each other. You can use zero or more whitespaces between the two literals. But you cannot mix an expression and a string literal. If the strings are inside parentheses, you can also use a newline character to separate the literals and optionally use comments.

```
>>> 'hello' r' 1\n2\\3'
'hello 1\\n2\\\\3'

# note that ... is REPL's indication for multiline statements, blocks, etc
>>> print('hi '
...       'there')
hi there
```

You can repeat a string by using the `*` operator between a string and an integer.

```
>>> style_char = '-'
>>> print(style_char * 50)
--------------------------------------------------
>>> word = 'buffalo '
>>> print(8 * word)
buffalo buffalo buffalo buffalo buffalo buffalo buffalo buffalo
```

String formatting

The Zen of Python (PEP 20) states:

> There should be one-- and preferably only one --obvious way to do it.

However, there are several approaches available for formatting strings. This section will first focus on **formatted** string literals (**f-strings** for short) and then show the alternate options.

f-strings allow you to embed an expression within `{}` characters as part of the string literal. Like raw strings, you need to use a prefix, which is `f` or `F` in this case. Python will substitute the embeds with the result of the expression, converting it to string if necessary (numeric results for example). See docs.python: Format String Syntax and docs.python: Formatted string literals for documentation and more examples.

```
>>> str1 = 'hello'
>>> str2 = ' world'
>>> f'{str1}{str2}'
'hello world'

>>> f'{str1}({str2 * 3})'
'hello( world world world)'
```

Use `{{` if you need to represent `{` literally. Similarly, use `}}` to represent `}` literally.

```
>>> f'{{hello'
'{hello'
>>> f'world}}'
'world}'
```

Adding `=` after an expression gives both the expression and the result in the output.

```
>>> num1 = 42
>>> num2 = 7

>>> f'{num1 + num2 = }'
'num1 + num2 = 49'
>>> f'{num1 + (num2 * 10) = }'
'num1 + (num2 * 10) = 112'
```

Optionally, you can provide a format specifier along with the expression after a `:` character. These specifiers are similar to the ones provided by the `printf()` function in **C** language, the `printf` built-in command in **Bash** and so on. Here are some examples for numeric formatting.

```
>>> appx_pi = 22 / 7

# restricting the number of digits after the decimal point
>>> f'Approx pi: {appx_pi:.5f}'
'Approx pi: 3.14286'

# rounding is applied
>>> f'{appx_pi:.3f}'
```

```
'3.143'

# exponential notation
>>> f'{32 ** appx_pi:.2e}'
'5.38e+04'
```

Here are some alignment examples:

```
>>> fruit = 'apple'

>>> f'{fruit:=>10}'
'=====apple'
>>> f'{fruit:=<10}'
'apple====='
>>> f'{fruit:=^10}'
'==apple==='

# default is the space character
>>> f'{fruit:^10}'
'  apple   '
```

You can use `b` , `o` and `x` to display integer values in binary, octal and hexadecimal formats respectively. Using `#` before these characters will add appropriate prefix for these formats.

```
>>> num = 42

>>> f'{num:b}'
'101010'
>>> f'{num:o}'
'52'
>>> f'{num:x}'
'2a'

>>> f'{num:#x}'
'0x2a'
```

The `str.format()` method, the `format()` function and the `%` operator are alternate approaches for string formatting.

```
>>> num1 = 22
>>> num2 = 7

>>> 'Output: {} / {} = {:.2f}'.format(num1, num2, num1 / num2)
'Output: 22 / 7 = 3.14'

>>> format(num1 / num2, '.2f')
'3.14'

>>> 'Approx pi: %.2f' % (num1 / num2)
'Approx pi: 3.14'
```

See docs.python: The String format() Method and the sections that follow for more details about the above features. See docs.python: Format examples for more examples, including `datetime` formatting. The Text processing chapter will discuss more about the string processing methods.

In case you don't know what a *method* is, see stackoverflow: What's the difference between a method and a function?

User input

The input() built-in function can be used to get data from the user. It also allows an optional string to make it an interactive process. This function always returns a string data type, which you can convert to another type if needed (explained in the next section).

```
# Python will wait until you type your text and press the Enter key
# the blinking cursor is represented by a rectangular block as shown below
>>> name = input('what is your name? ')
what is your name? █
```

Here's the rest of the above example.

```
>>> name = input('what is your name? ')
what is your name? learnbyexample

# note that newline isn't part of the value saved in the 'name' variable
>>> print(f'pleased to meet you {name}.')
pleased to meet you learnbyexample.
```

Type conversion

The type() built-in function can be used to know what data type you are dealing with. You can pass any expression as an argument.

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

>>> type(22 / 7)
<class 'float'>

>>> type('Hi there')
<class 'str'>
```

The built-in functions int(), float() and str() can be used to convert from one data type to another. These function names are the same as their data type class names seen above.

```
>>> num = 3.14
>>> int(num)
3
```

```
# you can also use f'{num}'
>>> str(num)
'3.14'

>>> usr_ip = input('enter a float value ')
enter a float value 45.24e22
>>> type(usr_ip)
<class 'str'>
>>> float(usr_ip)
4.524e+23
```

See docs.python: Built-in Functions for documentation on all of the built-in functions. You can also use the `help()` function from the REPL as discussed in the Documentation and getting help section.

Exercises

- Read about the **Bytes** literal from docs.python: String and Bytes literals. See also stackoverflow: What is the difference between a string and a byte string?
- If you check out docs.python: int() function, you'll see that the `int()` function accepts an optional argument. Write a program that asks the user for hexadecimal number as input. Then, use the `int()` function to convert the input string to an integer (you'll need the second argument for this). Add `5` and display the result in hexadecimal format.
- Write a program to accept two input values. First can be either a number or a string value. Second is an integer value, which should be used to display the first value in centered alignment. You can use any character you prefer to surround the value, other than the default space character.
- What happens if you use a combination of `r` , `f` and other such valid prefix characters while declaring a string literal? For example, `rf'a\{5/2}'` . What happens if you use the raw strings syntax and provide only a single `\` character? Does the documentation describe these cases?
- Try out at least two format specifiers not discussed in this chapter.
- Given `a = 5` , display `'{5}'` as the output using **f-strings**.

Defining functions

This chapter will discuss how to define your own functions, pass arguments to them and get back results. You'll also learn more about the `print()` built-in function.

def

Use the `def` keyword to define a function. The function name is specified after the keyword, followed by arguments inside parentheses and finally a `:` character to end the definition. It is a common mistake for beginners to miss the `:` character. Arguments are optional, as shown in the below program.

```python
# no_args.py
def greeting():
    print('-----------------------------')
    print('         Hello World         ')
    print('-----------------------------')

greeting()
```

The above code defines a function named `greeting` and contains three statements as part of the function. Unlike many other programming languages, whitespaces are significant in Python. Instead of a pair of curly braces, indentation is used to distinguish the body of the function and statements outside of that function. Typically, 4 space characters are used. The function call `greeting()` has the same indentation level as the function definition, so it is not part of the function. For readability purposes, an empty line has been used to separate the function definition and the subsequent statements.

```
$ python3.13 no_args.py
-----------------------------
         Hello World
-----------------------------
```

Functions have to be declared before they can be called. As an **exercise**, call the function before declaration and see what happens for the above program.

As per Style Guide for Python Code (PEP 8), it is recommended to use two blank lines around top level functions. However, I prefer to use a single blank line. For large projects, specialized tools like ruff are typically used to analyze and enforce coding styles/guidelines.

To create a placeholder function, one option is to use the `pass` statement to indicate no operation. See docs.python: pass statement for details.

Accepting arguments

Functions can accept one or more arguments separated by a comma.

```
# with_args.py
def greeting(ip):
    op_length = 10 + len(ip)
    styled_line = '-' * op_length
    print(styled_line)
    print(f'{ip:^{op_length}}')
    print(styled_line)

greeting('hi')
weather = 'Today would be a nice, sunny day'
greeting(weather)
```

In the above script, the function from the previous example has been modified to accept an input string as the sole argument. The len() built-in function is used here to get the length of a string value. The code also showcases the usefulness of variables, string operators and string formatting.

```
$ python3.13 with_args.py
------------
     hi
------------
------------------------------------------
     Today would be a nice, sunny day
------------------------------------------
```

As an **exercise**, modify the above program as suggested below and observe the results you get.

- add print statements for `ip` , `op_length` and `styled_line` variables at the end of the program (after the function calls)
- pass a numeric value to the `greeting()` function
- don't pass any argument while calling the `greeting()` function

The argument variables, and those that are defined within the body, are local to the function and would result in an exception if used outside the function. See also docs.python: Scopes and Namespaces and docs.python: global statement.

Python being a dynamically typed language, it is up to you to sanitize input for correctness. See also docs.python: Support for type hints and realpython: Python Type Checking Guide.

Default valued arguments

A default value can be specified during the function definition. Such arguments can be skipped during the function call, in which case they'll use the default value. They are also known as **keyword arguments**. Here's an example:

```python
# default_args.py
def greeting(ip, style='-', spacing=10):
    op_length = spacing + len(ip)
    styled_line = style * op_length
    print(styled_line)
    print(f'{ip:^{op_length}}')
    print(styled_line)

greeting('hi')
greeting('bye', spacing=5)
greeting('hello', style='=')
greeting('good day', ':', 2)
```

There are various ways in which you can call functions with default values. If you specify the argument name, they can be passed in any order. But, if you pass values positionally, the order has to be same as the declaration.

```
$ python3.13 default_args.py
------------
     hi
------------
--------
  bye
--------
===============
     hello
===============
::::::::::
 good day
::::::::::
```

As an **exercise**, modify the above script for the below requirements.

- make the spacing work for multicharacter `style` argument
- accept another argument with a default value of single space character that determines the character to be used around the centered `ip` value

As another **exercise**, what do you think will happen if you use `greeting(spacing=5, ip='Oh!')` to call the function shown above?

Arguments declared without default values can still be used as keyword arguments during function call. This is the default behavior. Python provides special constructs `/` and `*` for stricter separation of positional and keyword arguments. See docs.python: Special parameters for details.

Return value

The default return value of a function is `None`, which is typically used to indicate the absence of a meaningful value. The `print()` function, for example, has a `None` return value. Functions like `int()`, `len()` and `type()` have specific return values, as seen in prior examples.

```
>>> print('hi')
hi
>>> value = print('hi')
hi

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

Use the `return` statement to explicitly give back a value when the function is called. You can use this keyword by itself as well (default value is `None`).

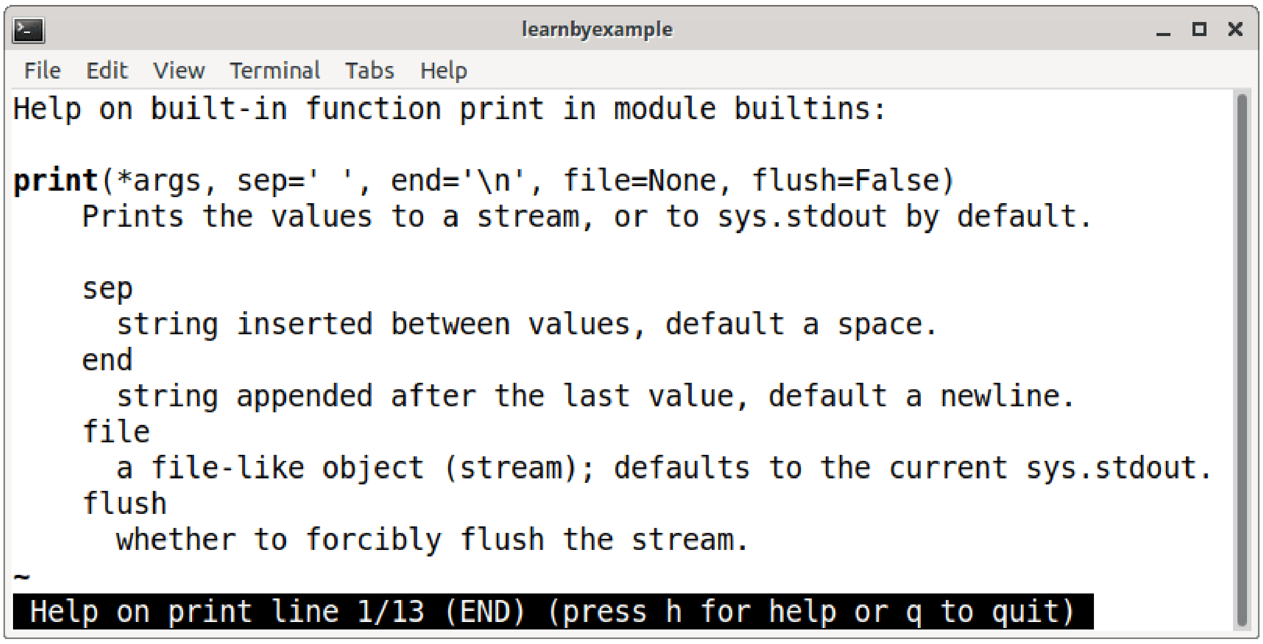
```
>>> def num_square(n):
...     return n * n
...
>>> num_square(5)
25
>>> num_square(3.14)
9.8596

>>> op = num_square(-42)
>>> type(op)
<class 'int'>
```

On encountering a `return` statement, the function will be terminated and further statements, if any, present as part of the function body will not be executed.

A common beginner confusion is mixing up the `print()` function and the `return` statement. See stackoverflow: What is the formal difference between ”print” and ”return”? for examples and explanations.

A closer look at the print() function

The `help` documentation for the `print()` function is shown below:



```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
~
 Help on print line 1/13 (END) (press h for help or q to quit)
```

As you can see, there are four default valued arguments. But, what does `*args` mean? It indicates that the `print()` function can accept arbitrary number of arguments.

```
# newline character is appended even if no arguments are passed
>>> print()

>>> print('hi')
hi
>>> print('hi', 5)
hi 5

>>> word1 = 'loaf'
>>> word2 = 'egg'
>>> print(word1, word2, 'apple roast nut')
loaf egg apple roast nut
```

If you observe closely, you'll notice that a **space** character is inserted between the arguments. That separator can be changed by using the `sep` argument.

```
>>> print('hi', 5, sep='')
hi5
>>> print('hi', 5, sep=':')
hi:5
>>> print('best', 'years', sep='.\n')
best.
years
```

Similarly, you can change the string that gets appended to something else.

```
>>> print('hi', end='----\n')
hi----
>>> print('hi', 'bye', sep='-', end='\n======\n')
hi-bye
======
```

The `file` argument will be discussed later. Writing your own function to accept arbitrary number of arguments will also be discussed later.

Docstrings

Triple quoted strings are also used for multiline comments and to document various part of a Python script. The latter is achieved by adding help content as string literals (but without being assigned to a variable) at the start of a function, class, etc. Such literals are known as documentation strings, or **docstrings** for short. Idiomatically, triple quoted strings are used for docstrings. The `help()` function reads these docstrings to display the documentation. There are also numerous third-party tools that make use of docstrings.

Here's an example:

```
>>> def num_square(n):
...     """
...     Returns the square of a number.
...     """
...     return n * n
...
>>> help(num_square)
```

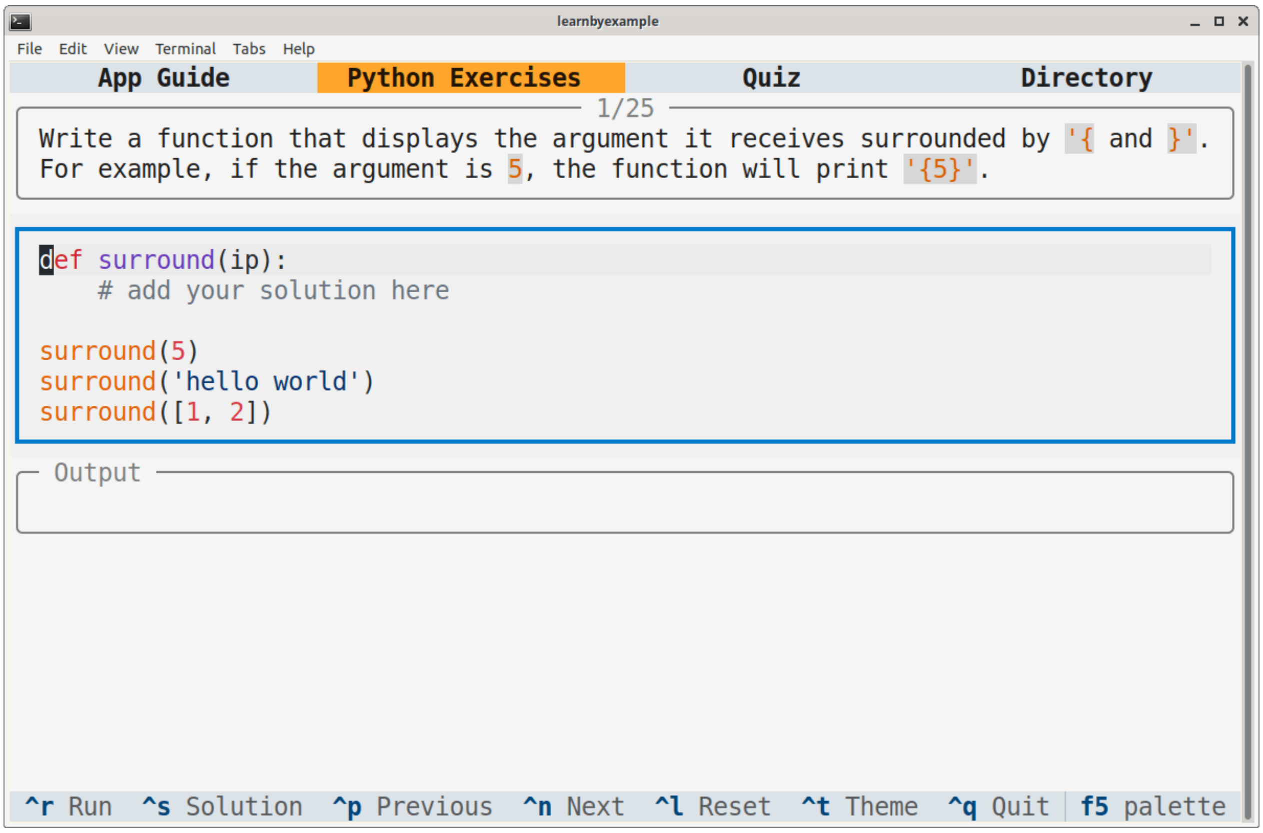
Calling `help(num_square)` will give you the documentation as shown below.

```
num_square(n)
    Returns the square of a number.
```

See docs.python: Documentation Strings for usage guidelines and other details.

Interactive TUI app for exercises

I wrote a TUI app that you can use to interactively solve most of the exercises from this book. See PythonExercises repo for installation instructions and usage guide. A sample screenshot is shown below:


learnbyexample
File Edit View Terminal Tabs Help
App Guide
Python Exercises
Quiz
Directory
1/25

Write a function that displays the argument it receives surrounded by '{ and }'.
For example, if the argument is 5, the function will print '{5}'.

```
def surround(ip):
    # add your solution here

surround(5)
surround('hello world')
surround([1, 2])
```

Output
^r Run ^s Solution ^p Previous ^n Next ^l Reset ^t Theme ^q Quit f5 palette

Control structures

This chapter shows operators used in conditional expressions, followed by control structures.

Comparison operators

These operators yield `True` or `False` boolean values as a result of comparison between two values.

```
>>> 0 != '0'
True
>>> 0 == int('0')
True
>>> 'hi' == 'Hi'
False

>>> 4 > 3.14
True
>>> 4 >= 4
True

>>> 'bat' < 'at'
False
>>> 2 <= 3
True
```

Python is a strictly typed language. So, unlike context-based languages like Perl, you have to explicitly use type conversion when needed. As an **exercise**, try using any of the `<` or `<=` or `>` or `>=` operators between numeric and string values.

See docs.python: Comparisons and docs.python: Operator precedence for documentation and other details.

Truthy and Falsy values

The values by themselves have *Truthy* and *Falsy* meanings when used in a conditional context. You can use the bool() built-in function to explicitly convert them to boolean values.

The numerical value **zero**, an **empty** string and `None` are *Falsy*. Non-zero numbers and non-empty strings are *Truthy*. See docs.python: Truth Value Testing for a complete list.

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>

>>> bool(4)
True
>>> bool(0)
False
```

```
>>> bool(-1)
True

>>> bool('')
False
>>> bool('hi')
True

>>> bool(None)
False
```

Boolean operators

You can use the `and` and `or` boolean operators to combine comparisons. The `not` operator is useful to invert a condition.

```
>>> 4 > 3.14 and 2 <= 3
True

>>> 'hi' == 'Hi' or 0 != '0'
True

>>> not 'bat' < 'at'
True
>>> num = 0
>>> not num
True
```

The `and` and `or` operators are also known as **short-circuit** operators. These will evaluate the second expression if and only if the first one evaluates to `True` and `False` respectively. Also, these operators return the result of the expressions used, which can be a non-boolean value. The `not` operator always returns a boolean value.

```
>>> num = 5
# here, num ** 2 will NOT be evaluated
>>> num < 3 and num ** 2
False
# here, num ** 2 will be evaluated as the first expression is True
>>> num < 10 and num ** 2
25
# not operator always gives a boolean value
>>> not (num < 10 and num ** 2)
False

>>> 0 or 3
3
>>> 1 or 3
1
```

Comparison chaining

Similar to mathematical notations, you can chain comparison operators. Apart from resulting in a terser conditional expression, this also has the advantage of having to evaluate the middle expression only once.

```
>>> num = 5

# using boolean operator
>>> num > 3 and num <= 5
True

# comparison chaining
>>> 3 < num <= 5
True
>>> 4 < num > 3
True
>>> 'bat' < 'cat' < 'cater'
True
```

Membership operator

The `in` comparison operator checks if a given value is part of a collection of values. Here's an example with the `range()` function:

```
>>> num = 5
# range() will be discussed in detail later in this chapter
# this checks if num is present among the integers 3 or 4 or 5
>>> num in range(3, 6)
True
>>> 6 in range(3, 6)
False
```

You can build your own collection of values using various data types like `list` , `set` , `tuple` etc. These data types will be discussed in detail in later chapters.

```
>>> num = 21
>>> num == 10 or num == 21 or num == 33
True
# RHS value here is a tuple data type
>>> num in (10, 21, 33)
True

>>> 'cat' not in ('bat', 'mat', 'pat', 'Cat')
True
```

When applied to strings, the `in` operator performs substring comparison.

```
>>> fruit = 'mango'
>>> 'an' in fruit
True
>>> 'at' in fruit
False
```

if-elif-else

Similar to the function definition, control structures require indenting its body of code. And, there's a `:` character after you specify the conditional expression. You should be already familiar with `if` and `else` keywords from other programming languages. Alternate conditional branches are specified using the `elif` keyword. You can nest these structures and each branch can have one or more statements.

Here's an example of an `if-else` structure within a user defined function. Note the use of indentation to separate different structures. Examples with the `elif` keyword will be seen later.

```python
# odd_even.py
def isodd(n):
    if n % 2:
        return True
    else:
        return False

print(f'{isodd(42) = }')
print(f'{isodd(-21) = }')
print(f'{isodd(123) = }')
```

Here's the output of the above program.

```
$ python3.13 odd_even.py
isodd(42) = False
isodd(-21) = True
isodd(123) = True
```

As an **exercise**, reduce the `isodd()` function body to a single statement instead of four. This is possible with features already discussed in this chapter — the ternary operator discussed in the next section would be an overkill.

Python doesn't support the `switch` control structure. See stackoverflow: switch statement in Python? for workarounds. docs.python: match statement is a powerful alternative to `switch`, introduced in the Python 3.10 version.

Ternary operator

Python doesn't support the traditional `?:` ternary operator syntax. Instead, it uses `if-else` keywords in the same line as illustrated below.

```python
def absolute(num):
    if num >= 0:
        return num
    else:
        return -num
```

The above `if-else` structure can be rewritten using the ternary operator as shown below:

```
def absolute(num):
    return num if num >= 0 else -num
```

Or, just use the abs() built-in function, which has support for complex numbers, fractions, etc. Unlike the above program, `abs()` will also handle `-0.0` correctly.

> See stackoverflow: ternary conditional operator for other ways to emulate the ternary operation in Python. `True` and `False` boolean values are equivalent to `1` and `0` in integer context. So, for example, the above ternary expression can also be written as `(-num, num)[num >= 0]` .

for loop

Counter based loop can be constructed using the range() built-in function and the `in` operator. The `range()` function can be called in the following ways:

```
range(stop)
range(start, stop)
range(start, stop, step)
```

Both ascending and descending order arithmetic progressions can be constructed using these variations. When skipped, the default values are `start=0` and `step=1` . For understanding purposes, a `C` -like code snippet is shown below:

```
# ascending order
for(i = start; i < stop; i += step)

# descending order
for(i = start; i > stop; i += step)
```

Here's a sample multiplication table:

```
>>> num = 9
>>> for i in range(1, 5):
...     print(f'{num} * {i} = {num * i}')
...
9 * 1 = 9
9 * 2 = 18
9 * 3 = 27
9 * 4 = 36
```

The `range` , `list` , `tuple` , `str` data types (and some more) fall under **sequence** types. There are multiple operations that are common to these types (see docs.python: Common Sequence Operations for details). For example, you could iterate over these types using the `for` loop. The `start:stop:step` slicing operation is another commonality among these types. You can test your understanding of the slicing syntax by converting a `range()` expression to `list` or `tuple` types.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

```
>>> list(range(2, 11, 2))
[2, 4, 6, 8, 10]

>>> list(range(120, 99, -4))
[120, 116, 112, 108, 104, 100]
```

As an **exercise**, create this arithmetic progression `-2, 1, 4, 7, 10, 13` using the `range()` function. Also, see what value you get during each iteration of `for c in 'hello'` .

while loop

Use `while` loop when you want to execute statements as long as the condition evaluates to `True` . Here's an example:

```
# countdown.py
count = int(input('Enter a positive integer: '))
while count > 0:
    print(count)
    count -= 1

print('Go!')
```

Here's a sample run of the above script:

```
$ python3.13 countdown.py
Enter a positive integer: 3
3
2
1
Go!
```

As an **exercise**, rewrite the above program using a `for` loop. Can you think of a scenario where you must use a `while` loop instead of `for` ?

Python doesn't support `++` or `--` operations. As shown in the above program, combining arithmetic operations with assignment is supported.

break and continue

The `break` statement is useful to quit the current loop immediately. Here's an example where you can keep getting the square root of a number until you enter an empty string. Recall that an empty string is *Falsy*.

```
>>> while True:
...     num = input('enter a number: ')
...     if not num:
...         break
...     print(f'square root of {num} is {float(num) ** 0.5:.4f}')
...
enter a number: 2
```

```
square root of 2 is 1.4142
enter a number: 3.14
square root of 3.14 is 1.7720
enter a number:
>>>
```

See also stackoverflow: breaking out of nested loops.

When `continue` is used, further statements are skipped and the next iteration of the loop is started, if any. For example, in file processing you often need to skip certain lines like headers, comments, etc.

```
>>> for num in range(10):
...     if num % 3:
...         continue
...     print(f'{num} * 2 = {num * 2}')
...
0 * 2 = 0
3 * 2 = 6
6 * 2 = 12
9 * 2 = 18
```

As an **exercise**, use appropriate `range()` logic so that the `if` statement is no longer needed.

See docs.python: break, continue, else for more details and the curious case of `else` clause in loops.

Assignment expression

Quoting from docs.python: Assignment expressions:

An assignment expression (sometimes also called a ”named expression” or ”walrus”) assigns an expression to an identifier, while also returning the value of the expression.

The `while` loop snippet from the previous section can be re-written using the assignment expression as shown below:

```
>>> while num := input('enter a number: '):
...     print(f'square root of {num} is {float(num) ** 0.5:.4f}')
...
enter a number: 2
square root of 2 is 1.4142
enter a number: 3.14
square root of 3.14 is 1.7720
enter a number:
>>>
```

See Assignment Expressions (PEP 572) and my book on regular expressions for more details and examples.

Exercises

- If you don't already know about **FizzBuzz**, check out the problem statement on rosettacode and implement it in Python. See also Why Can't Programmers.. Program?
- Print all numbers from `1` to `1000` (inclusive) which reads the same in reversed form in both the binary and decimal formats. For example, `33` in decimal is `100001` in binary and both of these are palindromic. You can either implement your own logic or search online for palindrome testing in Python.
- Write a function that returns the maximum nested depth of curly braces for a given string input. For example, `'{{a+2}*{{b+{c*d}}+e*d}}'` should give `4`. Unbalanced or wrongly ordered braces like `'{a}*b{'` and `'}a+b{'` should return `-1`.

If you'd like more exercises to test your understanding, check out these excellent resources:

- Exercism, Hackinscience and Practicepython — beginner friendly
- PythonExercises — my interactive TUI app
- Adventofcode, Codewars, Python Morsels — for intermediate to advanced level users
- Checkio, Codingame — gaming based challenges
- /r/dailyprogrammer — interesting challenges

See also Python Programming Exercises, Gently Explained — a free ebook that includes gentle explanations of the problem, the prerequisite coding concepts you'll need to understand the solution, etc.

Importing and creating modules

The previous chapters focused on data types, functions (both built-in and user defined) and control structures. This chapter will show how to use built-in as well as user defined modules. Quoting from docs.python: Modules:

> A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended.

Random numbers

Say you want to generate a random number from a given range for a guessing game. You could write your own random number generator. Or, you could save development/testing time, and make use of the random built-in module.

Here's an example guessing game:

```python
# rand_number.py
import random

# gives back an integer between 0 and 10 (inclusive)
rand_int = random.randrange(11)

print('I have thought of a number between 0 and 10.')
print('Can you guess it within 4 attempts?\n')
for _ in range(4):
    guess = int(input('Enter your guess: '))
    if guess == rand_int:
        print('Wow, you guessed it right!')
        break
    elif guess > rand_int:
        print('Oops, your guess is too high.')
    else:
        print('Oops, your guess is too low.')
else:
    print('\nOh no! You are out of chances. Better luck next time.')
```

`import random` indicates that you want to load this module. You'll see more details about `import` later in this chapter. The `randrange()` method follows the same `start/stop/step` logic as the `range()` function and returns a random integer from the given range. The `for` loop is used here to get the user input for a maximum of `4` attempts. The loop body doesn't need to know the current iteration count. In such cases, `_` is used to indicate a throwaway variable name.

As mentioned in the previous chapter, the `else` clause is supported by loops too. It is used to execute code if the loop is completed normally. If the user correctly guesses the random number, `break` will be executed, which is *not* a normal loop completion. In that case, the `else` clause will *not* be executed.

A sample run with correct guess is shown below.

```
$ python3.13 rand_number.py
I have thought of a number between 0 and 10
Can you guess it within 4 attempts?

Enter your guess: 5
Oops, your guess is too low.
Enter your guess: 8
Oops, your guess is too high.
Enter your guess: 6
Wow, you guessed it right!
```

Here’s a failed guess.

```
$ python3.13 rand_number.py
I have thought of a number between 0 and 10.
Can you guess it within 4 attempts?

Enter your guess: 1
Oops, your guess is too low.
Enter your guess: 2
Oops, your guess is too low.
Enter your guess: 3
Oops, your guess is too low.
Enter your guess: 4
Oops, your guess is too low.

Oh no! You are out of chances. Better luck next time.
```

Importing your own module

All the programs presented so far can be used as a module as it is without making further changes. However, that’ll lead to some unwanted behavior. This section will discuss these issues and the next section will show how to resolve them.

```python
# num_funcs.py
def sqr(n):
    return n * n

def fact(n):
    total = 1
    for i in range(2, n+1):
        total *= i
    return total

num = 5
print(f'square of {num} is {sqr(num)}')
print(f'factorial of {num} is {fact(num)}')
```

The above program defines two functions, one variable and calls the `print()` function twice. After you’ve written this program, open an interactive shell from the same directory. Then, load the module using `import num_funcs` where `num_funcs` is the name of the program without the `.py` extension.

```
>>> import num_funcs
square of 5 is 25
factorial of 5 is 120
```

So what happened here? Not only did the `sqr` and `fact` functions get imported, the code outside of these functions got executed as well. That isn't what you'd expect on loading a module. The next section will show how to prevent this behavior. For now, continue the REPL session.

```
>>> num_funcs.sqr(12)
144
>>> num_funcs.fact(0)
1
>>> num_funcs.num
5
```

As an **exercise**,

- add docstrings for the above program and check the output of the `help()` function using `num_funcs`, `num_funcs.fact`, etc as arguments.
- check what would be the output of `num_funcs.fact()` for negative integers and floating-point numbers. Then import the `math` built-in module and repeat the process with `math.factorial()`. Go through the Exception handling chapter and modify the above program to gracefully handle negative integers and floating-point numbers.

How does Python know where a module is located? Quoting from docs.python: The Module Search Path:

> When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in `sys.builtin_module_names`. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:
> • The directory containing the input script (or the current directory when no file is specified).
> • `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
> • The installation-dependent default (by convention including a `site-packages` directory, handled by the `site` module).

__name__ special variable

The special variable `__name__` will be assigned the *string* value `'__main__'` only for the program file that is executed. The files that are imported inside another program will see their own filename (without the extension) as the value for the `__name__` variable. This behavior allows you to code a program that can act as a module as well as execute extra code if and only if it is run as the main program.

Here's an example to illustrate this behavior:

```python
# num_funcs_module.py
def sqr(n):
    return n * n

def fact(n):
    total = 1
    for i in range(2, n+1):
        total *= i
    return total

if __name__ == '__main__':
    num = 5
    print(f'square of {num} is {sqr(num)}')
    print(f'factorial of {num} is {fact(num)}')
```

When you run the above program as a standalone application, the `if` condition will get evaluated to `True` .

```
$ python3.13 num_funcs_module.py
square of 5 is 25
factorial of 5 is 120
```

On importing, the above `if` condition will evaluate to `False` as `num_funcs_module.py` is no longer the main program. In the below example, the REPL session is the main program.

```python
>>> __name__
'__main__'
>>> import num_funcs_module
>>> num_funcs_module.sqr(12)
144
>>> num_funcs_module.fact(0)
1

# 'num' variable inside the 'if' block is no longer accessible
>>> num_funcs_module.num
Traceback (most recent call last):
  File "<python-input-4>", line 1, in <module>
    num_funcs_module.num
AttributeError: module 'num_funcs_module' has no attribute 'num'
```

In the above example, there are three statements that'll be executed if the program is run as the main program. It is common to put such statements under a `main()` user defined function and then call it inside the `if` block.

There are many such special variables and methods with **d**ouble **under**scores around their names. They are also called as **dunder** variables and methods. See stackoverflow: __name__ special variable for a detailed discussion and strange use cases.

Different ways of importing

When you use the `import <module>` statement, you'll have to prefix the module name whenever you need to use its features. If this becomes cumbersome, you can use alternate ways of importing.

First up, removing the prefix altogether as shown below. This will load all the names from the module except those beginning with a `_` character. Note that this method of importing a module is not recommended unless really necessary.

```
>>> from math import *
>>> sin(radians(90))
1.0
>>> pi
3.141592653589793
```

Instead of importing everything, a comma separated list of names is usually enough.

```
>>> from random import randrange
>>> randrange(3, 10, 2)
9

>>> from math import cos, pi
>>> cos(pi)
-1.0
```

You can also alias the name being imported using the `as` keyword. You can specify multiple aliases with comma separation.

```
>>> import random as rd
>>> rd.randrange(4)
1

>>> from math import factorial as fact
>>> fact(10)
3628800
```

__pycache__ directory

If you notice the `__pycache__` directory after you import your own module, **don't panic**. Quoting from docs.python: Compiled Python files:

> To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.version.pyc` , where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3 the compiled version of spam.py would be cached as `__pycache__/spam.cpython-33.pyc` . This naming convention allows compiled modules from different releases and different versions of Python to coexist.

You can use `python3.13 -B` if you do not wish the `__pycache__` directory to be created.

Explore modules

- docs.python: The Python Standard Library
- Awesome Python — a curated list of frameworks, libraries, software and resources
- Best of Python — a ranked list of open-source libraries and tools
- The Beginner’s Guide to Python Turtle and Step-by-Step Guide to Turtle Animations

Installing modules and Virtual environments

The standard modules are only a tiny fraction of the vast amount of libraries available for Python. The rich third-party ecosystem is one of the reasons why Python is so popular. You can find plenty of well made modules for web development, finance applications, scientific computing, machine learning, bioinformatics, data science, GUI, games, etc. There are plenty of alternatives for standard modules as well. Quoting from pypi.org: Python Packaging Index:

> The Python Package Index (PyPI) is a repository of software for the Python programming language. PyPI helps you find and install software developed and shared by the Python community.

This chapter will discuss how to use `pip` for installing modules. You'll also see how to create virtual environments using the `venv` module.

pip

Modern Python versions come with the `pip` installer program. The below code shows how to install the latest version of a module (along with dependencies, if any) from PyPI. See the pip user guide for documentation and other details like how to use `pip` on Windows. The `--user` option limits the availability of the module to the current user, see packaging.python: Installing to the User Site for more details.

```
# use py instead of python3.13 for Windows
$ python3.13 -m pip install --user regex
Collecting regex
  Downloading ...
Installing collected packages: regex
Successfully installed regex-2024.11.6
```

> ⚠ Make sure that the package to be installed supports your Python version.

Here's an example with the `regex` module that makes use of a subexpression call to recursively match nested sets of parentheses.

```
>>> import regex

>>> eqn = '((3+a) * ((r-2)*(t+2)/6) + 42 * (a(b(c(d(e)))))'
>>> regex.findall(r'\((?:[^()]++|(?0))++\)', eqn)
['(3+a)', '((r-2)*(t+2)/6)', '(a(b(c(d(e)))))']
```

> ℹ See packaging.python: Installing from PyPI for details like constraining package version number, upgrading packages, etc.

> ℹ `uninstall` instead of `install --user` in the above example will remove the package. See also stackoverflow: how to uninstall a package for details and gotchas.

⚠⚠⚠ Unless you really know what you are doing, do NOT ever use `pip` as an **admin** user. Problematic packages are an issue, see Malicious packages found to be typo-squatting and Hunting for Malicious Packages on PyPI for examples. See also security.stackexchange: PyPI security measures.

venv

Virtual environments allow you to work with specific Python and package versions without interfering with other projects. You can use the built-in module `venv` to easily create and manage virtual environments.

The flow I use is summarized below. If you are using an IDE, it will likely have options to create and manage virtual environments.

```
# this is needed only once
# 'new_project' is the name of the folder, can be new or already existing
$ python3.13 -m venv new_project

$ cd new_project/
$ source bin/activate
(new_project) $ # pip install <modules>
(new_project) $ # do some scripting
(new_project) $ deactivate
# you're now out of the virtual environment
```

Here, `new_project` is the name of the folder containing the virtual environment. If the folder doesn't already exist, a new folder will be created. The command `source bin/activate` enables the virtual environment. Among other things, `python` or `python3` will point to the version you used to create this environment, which is `python3.13` in the above example. The prompt will change to the name of the folder, which is an easy way to know that you are inside a virtual environment (unless your normal prompt is something similar). `pip install` will be restricted to this environment.

Once your work is done, use the `deactivate` command to exit the virtual environment. If you delete the folder, your installed modules in that environment will be lost as well.

ℹ The commands shown above will differ for Windows. See this article on Virtual Environments for details.

Here are some more resources about virtual environments:

- meribold: Virtual Environments Demystified
- calmcode.io: virtualenv
- realpython: Python Virtual Environments Primer
- stackoverflow: What is the difference between venv, pyvenv, pyenv, virtualenv, virtualenvwrapper, pipenv, etc?

Creating your own package

The packaging.python: Packaging Python Projects tutorial walks you through packaging a simple Python project. See also:

- stackoverflow: What is setup.py?
- Practical Python Programming: Packaging and Distribution
- How to create a Python package
- Comprehensive guide to Python project management and packaging

Exception handling

This chapter will discuss different types of errors and how to handle some of the them within the program gracefully. You'll also see how to raise exceptions programmatically.

Syntax errors

Quoting from docs.python: Errors and Exceptions:

> There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*

Here's an example program with syntax errors:

```
# syntax_error.py
print('hello')

def main():
    num = 5
    total = num + 09
    print(total)

main)
```

The above code is using an unsupported syntax for a numerical value. Note that the syntax check happens before any code is executed, which is why you don't see the output for the `print('hello')` statement. Can you spot the rest of the syntax issues in the above program?

```
$ python3.13 syntax_error.py
  File "/home/learnbyexample/Python/programs/syntax_error.py", line 5
    total = num + 09
                  ^
SyntaxError: leading zeros in decimal integer literals are not permitted;
             use an 0o prefix for octal integers
```

try-except

Exceptions happen when something goes wrong during the code execution. For example, passing a wrong data type to a function, dividing a number by `0` and so on. Such errors are typically difficult or impossible to determine just by looking at the code.

```
>>> int('42')
42
>>> int('42x')
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    int('42x')
    ~~~^^^^^^^
ValueError: invalid literal for int() with base 10: '42x'
>>> 3.14 / 0
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
```

```
    3.14 / 0
    ~~~~~^~~
ZeroDivisionError: float division by zero
```

When an exception occurs, the program stops executing and displays the line that caused the error. You also get an error type, such as `ValueError` and `ZeroDivisionError` seen in the above example, followed by a message. This may differ for user defined error types.

You can implement alternate code branches to be followed for certain types of errors instead of premature end to the program execution. For example, you could allow the user to correct their input data. In some cases, you want the program to end, but display a user friendly message instead of developer friendly traceback.

Put the code likely to generate an exception inside a `try` block and provide alternate paths inside one or more `except` blocks. Here's an example to get a positive integer number from the user, and continue doing so if the input was invalid.

```
# try_except.py
from math import factorial

while True:
    try:
        num = int(input('Enter a positive integer: '))
        print(f'{num}! = {factorial(num)}')
        break
    except ValueError:
        print('Not a positive integer, try again')
```

It so happens that both `int()` and `factorial()` generate `ValueError` in the above example. If you wish to take the same alternate path for multiple errors, you can pass a `tuple` to `except` instead of a single error type. Here's a sample run:

```
$ python3.13 try_except.py
Enter a positive integer: 3.14
Not a positive integer, try again
Enter a positive integer: hi
Not a positive integer, try again
Enter a positive integer: -2
Not a positive integer, try again
Enter a positive integer: 5
5! = 120
```

You can also capture the error message using the `as` keyword (which you have seen previously with the `import` statement, and discussed again in later chapters). Here's an example:

```
>>> try:
...     num = 5 / 0
... except ZeroDivisionError as e:
...     print(f'oops something went wrong! the error msg is:\n"{e}"')
...
oops something went wrong! the error msg is:
"division by zero"
```

See docs.python: built-in exceptions for documentation on built-in exceptions.

It is not recommended to use `except` without passing an error type. See stack-overflow: avoid bare exceptions for details. See also stackoverflow: get the name of an exception that was raised.

There are static code analysis tools like pylint, ”which looks for programming errors, helps enforcing a coding standard, sniffs for code smells and offers simple refactoring suggestions”. See awesome-python: code-analysis for more such tools.

else

The `else` clause behaves similarly to the `else` clause seen with loops. If there’s no exception raised in the `try` block, then the code in the `else` block will be executed. This block should be defined after the `except` blocks. As per the documentation:

> The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn’t raised by the code being protected by the `try ... except` statement.

```
# try_except_else.py
while True:
    try:
        num = int(input('Enter an integer number: '))
    except ValueError:
        print('Not an integer, try again')
    else:
        print(f'Square of {num} is {num ** 2}')
        break
```

Here’s a sample run:

```
$ python3.13 try_except_else.py
Enter an integer number: hi
Not an integer, try again
Enter an integer number: 3.14
Not an integer, try again
Enter an integer number: 42x
Not an integer, try again
Enter an integer number: -2
Square of -2 is 4
```

raise

You can manually `raise` exceptions if needed. It accepts an optional error type, which can be either a built-in or a user defined one (see docs.python: User-defined Exceptions). And you can optionally specify an error message. `raise` by itself re-raises the currently active exception, if any (`RuntimeError` otherwise).

```
>>> def sum2nums(n1, n2):
...     types_allowed = (int, float)
...     if type(n1) not in types_allowed or type(n2) not in types_allowed:
...         raise TypeError('Argument should be an integer or a float value')
...     return n1 + n2
...
>>> sum2nums(3.14, -2)
1.1400000000000001
>>> sum2nums(3.14, 'a')
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    sum2nums(3.14, 'a')
    ~~~~~~~~^^^^^^^^^^^
  File "<python-input-0>", line 4, in sum2nums
    raise TypeError('Argument should be an integer or a float value')
TypeError: Argument should be an integer or a float value
```

finally

You can add code in the `finally` block that should always be the last thing done by the `try` statement, irrespective of whether an exception has occurred. This should be declared after `except` and the optional `else` blocks.

```
# try_except_finally.py
try:
    num = int(input('Enter a positive integer: '))
    if num < 0:
        raise ValueError
except ValueError:
    print('Not a positive integer, run the program again')
else:
    print(f'Square root of {num} is {num ** 0.5:.3f}')
finally:
    print('\nThanks for using the program, have a nice day')
```

Here are some sample runs when the user enters some value:

```
$ python3.13 try_except_finally.py
Enter a positive integer: -2
Not a positive integer, run the program again

Thanks for using the program, have a nice day
$ python3.13 try_except_finally.py
Enter a positive integer: 2
Square root of 2 is 1.414
```

```
Thanks for using the program, have a nice day
```

Here’s an example where something goes wrong, but not handled by the `try` statement. Note that `finally` block is still executed.

```
# here, user presses Ctrl+D instead of entering a value
# you'll get KeyboardInterrupt if the user presses Ctrl+C
$ python3.13 try_except_finally.py
Enter a positive integer:
Thanks for using the program, have a nice day
Traceback (most recent call last):
  File "/home/learnbyexample/Python/programs/try_except_finally.py",
      line 2, in <module>
    num = int(input('Enter a positive integer: '))
              ~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
EOFError
```

See docs.python: Defining Clean-up Actions for details like what happens if an exception occurs within an `else` clause, presence of `break/continue/return` etc. The documentation also gives examples of where `finally` is typically used.

Exercises

- Identify the syntax errors in the following code snippets. Try to spot them manually.

  ```python
  # snippet 1:
  def greeting()
      print('hello')

  # snippet 2:
  num = 5
  if num = 4:
      print('what is going on?!')

  # snippet 3:
  greeting = “hi”
  ```

- In case you didn’t complete the exercises from the Importing your own module section, you should be able to do it now.

- Write a function `num(ip)` that accepts a single argument and returns the corresponding integer or floating-point number contained in the argument. Only `int` , `float` and `str` should be accepted as a valid input data type. Provide custom error messages if the input cannot be converted to a valid number. Examples are shown below:

  ```python
  >>> num(0x1f)
  31
  >>> num(3.32)
  3.32
  >>> num(' \t 52 \t')
  52
  ```

```
>>> num('3.982e5')
398200.0

# wrong data type
>>> num(['1', '2.3'])
TypeError: not a valid input
# string input that cannot be converted to a valid int/float number
>>> num('foo')
ValueError: could not convert string to int or float
```

Debugging

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. — Brian W. Kernighan

There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors. — Leon Bambrick

Debuggers don't remove bugs. They only show them in slow motion. — Unknown

These quotes were chosen from this collection at softwareengineering.stackexchange.

General tips

Knowing how to debug your programs is crucial and should be ideally taught right from the start instead of a chapter at the end of a beginner's learning resource. Think Python is an awesome example for such a resource material.

Debugging is often a frustrating experience. Taking a break helps. It is common to find or solve issues in your dreams too (I've had my share of these, especially during college and intense work days).

If you are stuck with a problem, reduce the code as much as possible so that you are left with the minimal code necessary to reproduce the issue. Talking about the problem to a friend/colleague/inanimate-objects/etc can help too — famously termed as Rubber duck debugging. I have often found the issue while formulating a question to be asked on forums like stackoverflow/reddit because writing down your problem is another way to bring clarity than just having a vague idea in your mind.

Here are some awesome articles on this challenging topic:

- What does debugging a program look like?
- How to debug small programs
- Debugging guide
- Problem solving skills

Here's a summarized snippet from a collection of interesting bug stories:

A jpeg parser choked whenever the CEO came into the room, because he always had a shirt with a square pattern on it, which triggered some special case of contrast and block boundary algorithms.

See also this curated list of absurd software bug stories.

Common beginner mistakes

The previous chapter already covered examples for syntax errors. This section will discuss some more Python gotchas.

Python allows you to redefine built-in functions, modules, classes, etc (see also stackoverflow: metaprogramming). Unless that's your intention, do *not* use keywords, built-in functions and modules as your variable name, function name, program filename, etc. Here's an example:

```
# normal behavior
>>> str(2)
'2'

# unintentional use of 'str' as a variable name
>>> str = input("what is your name? ")
what is your name? learnbyexample
# 'str' is now no longer usable as a built-in function
>>> str(2)
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    str(2)
    ~~~^^^
TypeError: 'str' object is not callable
```

Here's another example:

```
>>> len = 5
>>> len('hi')
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    len('hi')
    ~~~^^^^^^
TypeError: 'int' object is not callable
```

As an **exercise**, create an empty file named as `math.py`. In the same directory, create another program file that imports the `math` module and then uses some feature, `print(math.pi)` for example. What happens if you execute this program?

See also:

- Think Python: Debugging chapter
- inventwithpython: common runtime errors and common python gotchas
- pythonforbiologists: common beginner errors
- stackoverflow: common pitfalls in Python

pdb

Python comes with a handy built-in library `pdb` that you can use from the CLI to debug programs. See docs.python: pdb for documentation. Here are some of the frequently used commands (only short form is shown, see documentation for long form and more details).

- `l` prints code around the current statement the debugger is at, useful to visualize the progress of the debug effort
- `ll` prints the full code for the current function or frame
- `s` execute the current line, steps inside function calls
- `n` execute the current line, treats function call as a single execution step
- `c` continue execution until the next breakpoint

- `p expression` print value of an expression in the current context, usually used to see the current value of a variable
- `h` list of available commands
    - `h c` help on the `c` command
- `q` quit the debugger

Here's an example invocation of the debugger for the `num_funcs.py` program seen earlier in the Importing your own module section. Only the `n` command is used below. Lines with the `>` prefix tells you about the program file being debugged, current line number, function name and return value when applicable. Lines with the `->` prefix is the code present at the current line. `(Pdb)` is the prompt for this interactive session. You can also see the output of `print()` function for the last `n` command in the illustration below.

```
$ python3.13 -m pdb num_funcs.py
> /home/learnbyexample/Python/programs/num_funcs.py(1)<module>()
-> def sqr(n):
(Pdb) n
> /home/learnbyexample/Python/programs/num_funcs.py(4)<module>()
-> def fact(n):
(Pdb) n
> /home/learnbyexample/Python/programs/num_funcs.py(10)<module>()
-> num = 5
(Pdb) n
> /home/learnbyexample/Python/programs/num_funcs.py(11)<module>()
-> print(f'square of {num} is {sqr(num)}')
(Pdb) n
square of 5 is 25
> /home/learnbyexample/Python/programs/num_funcs.py(12)<module>()
-> print(f'factorial of {num} is {fact(num)}')
```

Continuation of the above debugging session is shown below, this time with the `s` command to step into the function. Use `r` while you are still inside the function to skip until the function encounters a `return` statement. Examples for the `p` and `ll` commands are also shown below.

```
(Pdb) s
--Call--
> /home/learnbyexample/Python/programs/num_funcs.py(4)fact()
-> def fact(n):
(Pdb) ll
  4  -> def fact(n):
  5         total = 1
  6         for i in range(2, n+1):
  7             total *= i
  8         return total
(Pdb) n
> /home/learnbyexample/Python/programs/num_funcs.py(5)fact()
-> total = 1
(Pdb) p n
5
(Pdb) r
```

```
--Return--
> /home/learnbyexample/Python/programs/num_funcs.py(8)fact()->120
-> return total
(Pdb) n
factorial of 5 is 120
--Return--
> /home/learnbyexample/Python/programs/num_funcs.py(12)<module>()->None
-> print(f'factorial of {num} is {fact(num)}')
```

Use `q` to end the session.

```
(Pdb) n
--Return--
> <string>(1)<module>()->None
(Pdb) q
```

You can call breakpoint() or pdb.set_trace() to set breakpoints in the code and use it in combination with the `c` command.

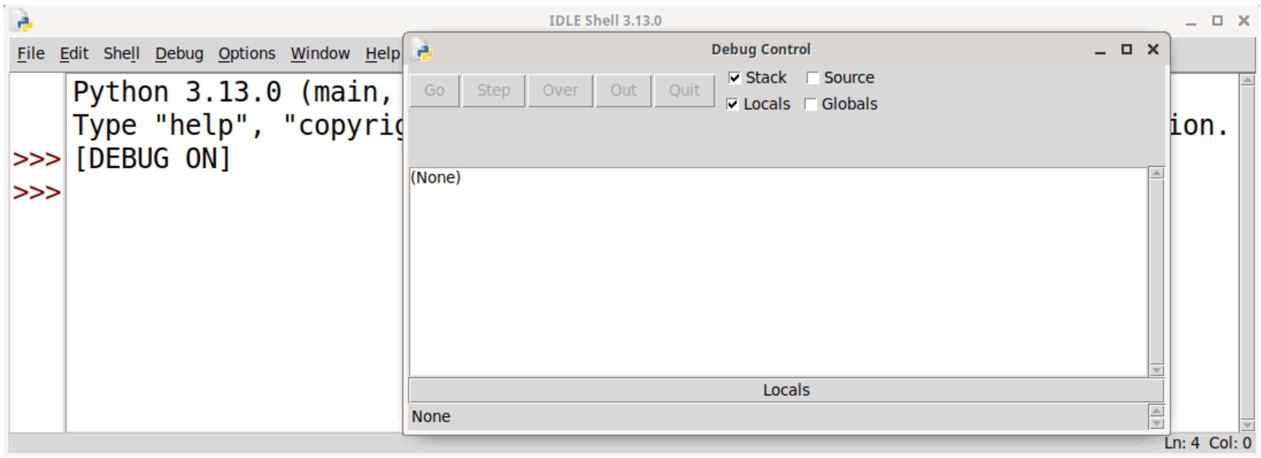
See also:

- awesome-python: Debugging tools
- pdb tutorial
- docs.python HOWTOs: Basic Logging Tutorial

IDLE debugging

Sites like Pythontutor allow you to visually debug a program — you can execute a program step by step and see the current value of variables. Similar features are typically provided by IDEs as well. Under the hood, these visualizations would likely be using the `pdb` module discussed in the previous section.

This section will show an example with `IDLE`. Before you can run the program, first select the **Debugger** option under the **Debug** menu. You can also use `idle3.13 -d` to launch IDLE in debug mode directly. You'll see a new window pop up as shown below:

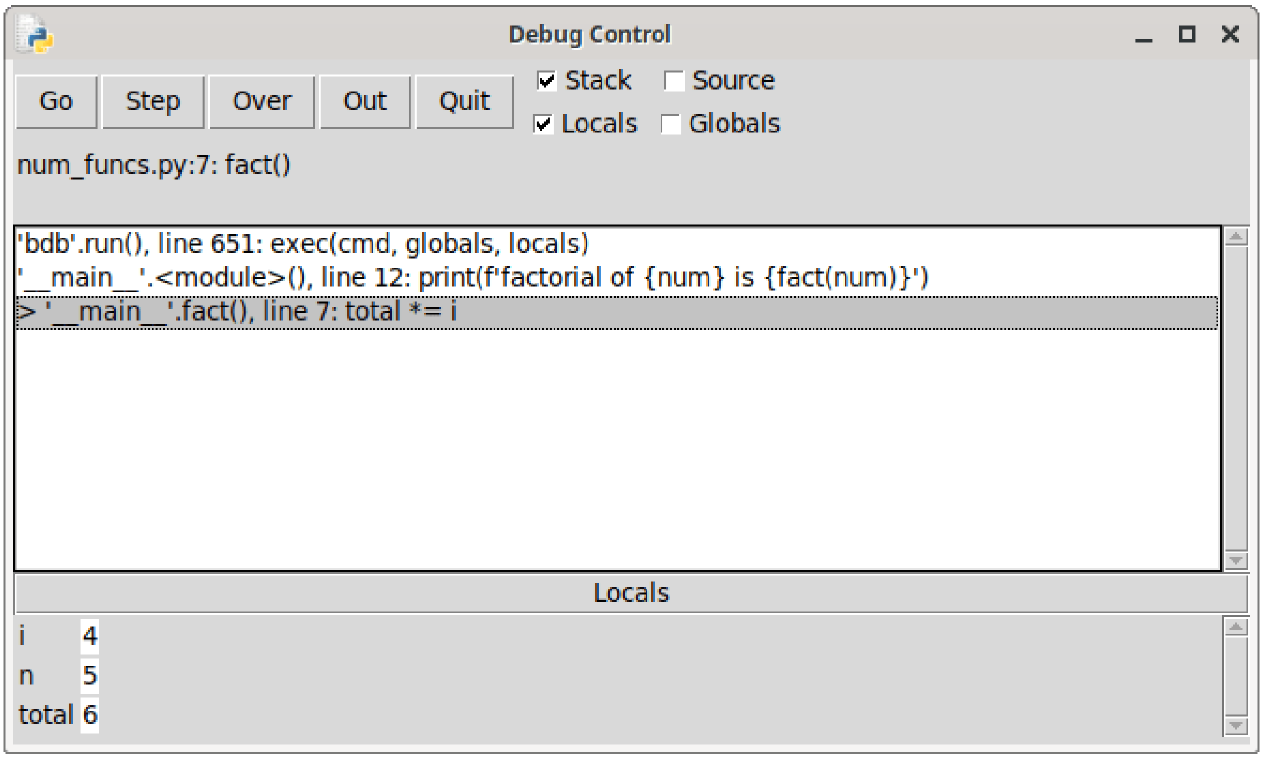


```
Python 3.13.0 (main,
Type "help", "copyrig
>>> [DEBUG ON]
>>>
```

(None)

None

Then, with debug mode active, load and run a program (for example, `num_funcs.py` discussed in the previous section). Use the buttons and options to go over the code. Variable values will be automatically available, as shown below.



```
num_funcs.py:7: fact()

'bdb'.run(), line 651: exec(cmd, globals, locals)
'__main__'.<module>(), line 12: print(f'factorial of {num} is {fact(num)}')
> '__main__'.fact(), line 7: total *= i

Locals
i     4
n     5
total 6
```

You can right-click on a line from the text editor to set or clear breakpoints.

See realpython: Debug With IDLE for a more detailed tutorial.

Testing

Testing can only prove the presence of bugs, not their absence. — Edsger W. Dijkstra

There, it should work now. — All programmers

These quotes were chosen from this collection at softwareengineering.stackexchange.

General tips

Another crucial aspect in the programming journey is knowing how to write tests. In bigger projects, usually there are separate engineers (often in much larger number than developers) to test the code. Even in those cases, writing a few sanity test cases yourself can help you develop faster knowing that the changes aren't breaking basic functionality.

There's no single consensus on test methodologies. There is Unit testing, Integration testing, Test-driven development (TDD) and so on. Often, a combination of these is used. These days, machine learning is also being used to reduce the testing time, see Testing Firefox more efficiently with machine learning for an example.

When I start a project, I usually try to write the programs incrementally. Say I need to iterate over files from a directory. I will make sure that portion is working (usually with `print()` statements), then add another feature — say file reading and test that and so on. This reduces the burden of testing a large program at once at the end. And depending upon the nature of the program, I'll add a few sanity tests at the end. For example, for my command_help project, I copy-pasted a few test runs of the program with different options and arguments into a separate file and wrote a program to perform these tests programmatically whenever the source code was modified.

assert

For simple cases, the `assert` statement is good enough. If the expression passed to `assert` evaluates to `False`, the `AssertionError` exception will be raised. You can optionally pass a message, separated by a comma after the expression to be tested. See docs.python: assert for documentation.

```
# passing case
>>> assert 2 < 3

# failing case
>>> num = -2
>>> assert num >= 0, 'only positive integer allowed'
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    assert num >= 0, 'only positive integer allowed'
           ^^^^^^^^
AssertionError: only positive integer allowed
```

Consider this sample program (solution for one of the exercises from the Control structures chapter).

```
# nested_braces.py
def max_nested_braces(expr):
    max_count = count = 0
    for char in expr:
        if char == '{':
            count += 1
            if count > max_count:
                max_count = count
        elif char == '}':
            if count == 0:
                return -1
            count -= 1

    if count != 0:
        return -1
    return max_count

def test_cases():
    assert max_nested_braces('a*b') == 0
    assert max_nested_braces('a*b+{}') == 1
    assert max_nested_braces('a*{b+c}') == 1
    assert max_nested_braces('{a+2}*{b+c}') == 1
    assert max_nested_braces('a*{b+c*{e*3.14}}') == 2
    assert max_nested_braces('{{a+2}*{b+c}+e}') == 2
    assert max_nested_braces('{{a+2}*{b+{c*d}}+e}') == 3
    assert max_nested_braces('{{a+2}*{{b+{c*d}}+e*d}}') == 4
    assert max_nested_braces('a*b{') == -1
    assert max_nested_braces('a*{b+c}}') == -1
    assert max_nested_braces('}a+b{') == -1
    assert max_nested_braces('a*{b+c*{e*3.14}}}') == -1
    assert max_nested_braces('{{a+2}*{{b}+{c*d}}+e*d}}') == -1

if __name__ == '__main__':
    test_cases()
    print('all tests passed')
```

`max_count = count = 0` is a terse way to initialize multiple variables to the same value. Okay to use for immutable types like `int`, `float` and `str`. See the Mutability chapter for more details.

If everything goes right, you should see the following output.

```
$ python3.13 nested_braces.py
all tests passed
```

As an **exercise**, randomly change the logic of the `max_nested_braces()` function and see if any of the tests fail.

The `assert` statements can be skipped if you use `python3.13 -O <filename>` to execute the script.

Writing tests helps you in many ways. It could help you guard against typos and accidental editing. Often, you'll need to tweak a program in future to correct some bugs or add a feature — tests would again help to ascertain that you haven't messed up already working cases. Another use case is **refactoring**, where you rewrite a portion of the program without changing its functionality.

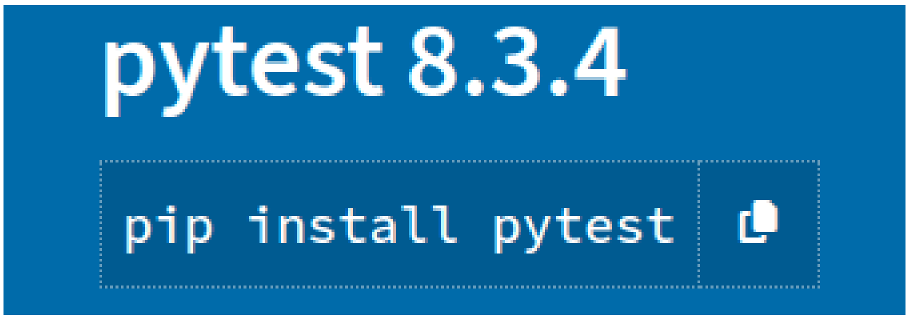
Here's an alternate implementation of the `max_nested_braces(expr)` function from the above program using regular expressions.

```python
# nested_braces_re.py
# only the function is shown below
import re

def max_nested_braces(expr):
    count = 0
    while True:
        expr, no_of_subs = re.subn(r'\{[^{}]*\}', '', expr)
        if no_of_subs == 0:
            break
        count += 1

    if re.search(r'[{}]', expr):
        return -1
    return count
```

pytest

For larger projects, simple `assert` statements aren't enough to adequately write and manage tests. You'll require built-in module unittest or popular third-party modules like pytest. See python test automation frameworks for more resources.

This section will show a few introductory examples with `pytest`. If you visit a project on PyPI, the pytest page for example, you can copy the installation command as shown in the image below. You can also check out the statistics link (https://libraries.io/pypi/pytest for example) as a minimal sanity check that you are installing the correct module.



```
pip install pytest
```

```
# virtual environment
$ pip install pytest
```

```
# normal environment
$ python3.13 -m pip install --user pytest
```

After installation, you'll have `pytest` usable as a command line application by itself. The two programs discussed in the previous section can be run without any modification as shown below. This is because `pytest` will automatically use function names starting with `test` for its purpose. See doc.pytest: Conventions for Python test discovery for full details.

```
# -v is the verbose option, use -q for the quiet version
$ pytest -v nested_braces.py
=================== test session starts ====================
platform linux -- Python 3.13.0, pytest-8.3.4,
    pluggy-1.5.0 -- /usr/local/bin/python3.13
cachedir: .pytest_cache
rootdir: /home/learnbyexample/Python/programs
collected 1 item

nested_braces.py::test_cases PASSED                  [100%]

==================== 1 passed in 0.01s =====================
```

Here's an example where `pytest` is used within the script itself.

```python
# exception_testing.py
import pytest

def sum2nums(n1, n2):
    types_allowed = (int, float)
    assert type(n1) in types_allowed, 'only int/float allowed'
    assert type(n2) in types_allowed, 'only int/float allowed'
    return n1 + n2

def test_valid_values():
    assert sum2nums(3, -2) == 1
    # see https://stackoverflow.com/q/5595425
    from math import isclose
    assert isclose(sum2nums(-3.14, 2), -1.14)

def test_exception():
    with pytest.raises(AssertionError) as e:
        sum2nums('hi', 3)
    assert 'only int/float allowed' in str(e.value)

    with pytest.raises(AssertionError) as e:
        sum2nums(3.14, 'a')
    assert 'only int/float allowed' in str(e.value)
```

`pytest.raises()` allows you to check if exceptions are raised for the given test cases. You can optionally check the error message as well. The `with` context manager will be discussed in a later chapter. Note that the above program doesn't actually call any executable code, since `pytest` will automatically run the test functions.

```
$ pytest -v exception_testing.py
=================== test session starts ====================
platform linux -- Python 3.13.0, pytest-8.3.4,
    pluggy-1.5.0 -- /usr/local/bin/python3.13
cachedir: .pytest_cache
rootdir: /home/learnbyexample/Python/programs
collected 2 items

exception_testing.py::test_valid_values PASSED       [ 50%]
exception_testing.py::test_exception PASSED          [100%]

==================== 2 passed in 0.01s =====================
```

The above illustrations are trivial examples. And tests are typically organized in different files/folders from the scripts being tested. Here are some advanced learning resources:

- Python testing style guide
- realpython: Getting started with testing in Python
- `pytest` — calmcode video series and Testing Python Applications
- obeythetestinggoat — TDD for the Web, with Python, Selenium, Django, JavaScript and pals
- testdriven: Modern Test-Driven Development in Python — TDD guide and has a real world application example
- Serious Python — deployment, scalability, testing, and more

Tuple and Sequence operations

This chapter will discuss the `tuple` data type and some of the common sequence operations. Data types like `str` , `range` , `list` and `tuple` fall under **Sequence** types.

Sequences and iterables

Quoting from docs.python glossary: **sequence**:

> An iterable which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list` , `str` , `tuple` , and `bytes` . Note that dict also supports `__getitem__()` and `__len__()` , but is considered a mapping rather than a sequence because the lookups use arbitrary hashable keys rather than integers.

Partial quote from docs.python glossary: **iterable**:

> An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list` , `str` , and `tuple`) and some non-sequence types like `dict` , file objects...

Some of the operations behave differently or do not apply for certain types, see docs.python: Common Sequence Operations for details.

Initialization

Tuples are declared as a collection of zero or more objects, separated by a comma within parentheses `()` characters. Each element can be specified as a value by itself or as an expression. The outer parentheses are optional if comma separation is present. Here are some examples:

```
# can also use: empty_tuple = tuple()
>>> empty_tuple = ()

# note the trailing comma, otherwise it will result in a 'str' data type
# same as 'apple', since parentheses are optional here
>>> one_element = ('apple',)

# multiple elements
>>> dishes = ('Aloo tikki', 'Baati', 'Khichdi', 'Makki roti', 'Poha')

# mixed data type example, uses expressions as well
>>> mixed = (1+2, 'two', (-3, -4), empty_tuple)
>>> mixed
(3, 'two', (-3, -4), ())
```

You can use the tuple() built-in function to create a `tuple` from an *iterable* (described in the previous section).

```
>>> chars = tuple('hello')
>>> chars
('h', 'e', 'l', 'l', 'o')
>>> tuple(range(3, 10, 3))
(3, 6, 9)
```

Tuples are immutable, but individual elements can be either mutable or immutable. As an **exercise**, given `chars = tuple('hello')` , check the output of the expression `chars[0]` and the statement `chars[0] = 'H'` .

Slicing

One or more elements can be retrieved from a sequence using the slicing notation (this wouldn't work for an iterable like `dict` or `set`). It works similar to the `start/stop/step` logic seen with the `range()` function. The default `step` is `1` . Default value for `start` and `stop` depends on whether the `step` is positive or negative.

```
>>> primes = (2, 3, 5, 7, 11)

# index starts with 0
>>> primes[0]
2

# start=2 and stop=4, default step=1
# note that the element at index 4 (stop value) isn't part of the output
>>> primes[2:4]
(5, 7)
# default start=0
>>> primes[:3]
(2, 3, 5)
# default stop=len(seq) for positive values of step
>>> primes[3:]
(7, 11)

# shallow copy of the sequence, same as primes[::1]
>>> primes[:]
(2, 3, 5, 7, 11)
```

You can use negative indexing to get elements from the end of the sequence. This is especially helpful when you don't know the size of the sequence. Given a positive integer `n` greater than zero, the expression `seq[-n]` is evaluated as `seq[len(seq) - n]` .

```
>>> primes = (2, 3, 5, 7, 11)

# len(primes) - 1 = 4, so this is same as primes[4]
>>> primes[-1]
11
```

```
# seq[-n:] will give the last n elements
>>> primes[-1:]
(11,)
>>> primes[-2:]
(7, 11)
```

Here are some examples with different `step` values.

```
>>> primes = (2, 3, 5, 7, 11)

# same as primes[0:5:2]
>>> primes[::2]
(2, 5, 11)

# retrieve elements in the reverse direction
# note that the element at index 1 (stop value) isn't part of the output
>>> primes[3:1:-1]
(7, 5)
# reversed sequence
# would help you with the palindrome exercise from Control structures chapter
>>> primes[::-1]
(11, 7, 5, 3, 2)
```

As an **exercise**, given `primes = (2, 3, 5, 7, 11)` ,

- what happens if you use `primes[5]` or `primes[-6]` ?
- what happens if you use `primes[:5]` or `primes[-6:]` ?
- is it possible to get the same output as `primes[::-1]` by using an explicit number for the `stop` value? If not, why not?

Sequence unpacking

You can assign the individual elements of an iterable to multiple variables. This is known as **sequence unpacking** and it is handy in many situations.

```
>>> details = ('2024-10-25', 'car', 2346)
>>> purchase_date, vehicle, qty = details
>>> purchase_date
'2024-10-25'
>>> vehicle
'car'
>>> qty
2346
```

Here's how you can easily swap variable values.

```
>>> num1 = 3.14
>>> num2 = 42
>>> num3 = -100

# RHS is a single tuple object (recall that parentheses are optional)
>>> num1, num2, num3 = num3, num1, num2
>>> print(f'{num1 = }; {num2 = }; {num3 = }')
num1 = -100; num2 = 3.14; num3 = 42
```

Unpacking isn't limited to single value assignments. You can use a `*` prefix to assign all the remaining values, if any is left, to a `list` variable.

```
>>> values = ('first', 6.2, -3, 500, 'last')

>>> x, *y = values
>>> x
'first'
>>> y
[6.2, -3, 500, 'last']

>>> a, *b, c = values
>>> a
'first'
>>> b
[6.2, -3, 500]
>>> c
'last'
```

As an **exercise**, what do you think will happen for these cases, given `nums = (1, 2)` :

- `a, b, c = nums`
- `a, *b, c = nums`
- `*a, *b = nums`

Returning multiple values

Tuples are also the preferred way to return multiple values from a function. Here's an example:

```
>>> def min_max(iterable):
...     return min(iterable), max(iterable)
...
>>> min_max('visualization')
('a', 'z')
>>> small, big = min_max((10, -42, 53.2, -3))
>>> small
-42
>>> big
53.2
```

The `min_max(iterable)` user-defined function in the above snippet returns both the **minimum** and **maximum** values of a given iterable input. `min()` and `max()` are built-in functions. You can either save the output as a `tuple` or unpack into multiple variables. You'll see built-in functions that return a `tuple` as output later in this chapter.

⚠️ The use of both min() and max() in the above example is for illustration purposes only. As an **exercise**, write a custom logic that iterates only once over the input sequence and calculates both the minimum and maximum values simultaneously.

Iteration

You have already seen examples of `for` loops that iterate over a sequence data type. Here's a refresher:

```
>>> nums = (3, 6, 9)
>>> for n in nums:
...     print(f'square of {n} is {n ** 2}')
...
square of 3 is 9
square of 6 is 36
square of 9 is 81
```

In the above example, you get one element per each iteration. If you need the **index** of the elements as well, you can use the enumerate() built-in function. You'll get a `tuple` value per each iteration, containing the index (starting with `0` by default) and the value at that index. Here are some examples:

```
>>> nums = (42, 3.14, -2)
>>> for t in enumerate(nums):
...     print(t)
...
(0, 42)
(1, 3.14)
(2, -2)
>>> for idx, val in enumerate(nums):
...     print(f'{idx}: {val:>5}')
...
0:    42
1:  3.14
2:    -2
```

ℹ️ The `enumerate()` built-in function has a `start=0` default valued argument. As an **exercise**, change the above snippet to start the index from `1` instead of `0` .

Arbitrary number of arguments

As seen before, the `print()` function can accept zero or more values separated by a comma. Here's a portion of the documentation as a refresher:

```
print(*args, sep=' ', end='\n', file=None, flush=False)
```

You can write your own functions to accept arbitrary number of arguments as well. The packing syntax is similar to the sequence unpacking examples seen earlier. A `*` prefix to an argument name will allow it to accept zero or more values. Such an argument will be packed as a `tuple` data type and it should always be specified after positional arguments (if any). Idiomatically, `args` is used as the variable name. Here's an example:

```
>>> def many(a, *args):
...     print(f'{a = }; {args = }')
...
>>> many()
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    many()
    ~~~~^^
TypeError: many() missing 1 required positional argument: 'a'
>>> many(1)
a = 1; args = ()
>>> many(1, 'two', 3)
a = 1; args = ('two', 3)
```

Here's a more practical example:

```
>>> def sum_nums(*args):
...     total = 0
...     for n in args:
...         total += n
...     return total
...
>>> sum_nums()
0
>>> sum_nums(3, -8)
-5
>>> sum_nums(1, 2, 3, 4, 5)
15
```

As an **exercise**,

- add a default valued argument `initial` which should be used to initialize `total` instead of `0` for the `sum_nums()` function. For example, `sum_nums(3, -8)` should give `-5` and `sum_nums(1, 2, 3, 4, 5, initial=5)` should give `20` .
- what would happen if you use `sum_nums(initial=5, 2)` to call this function?
- what would happen if you have `nums = (1, 2)` and use `sum_nums(*nums, initial=3)` to call the function?
- in what ways does this function differ from the sum() built-in function?

See also docs.python: Arbitrary Argument Lists.

The Arbitrary keyword arguments section in a later chapter will discuss how to define functions that accept arbitrary number of keyword arguments.

zip

You can use zip() to iterate over two or more iterables simultaneously. Every iteration, you'll get a `tuple` with an item from each of the iterables. Here's an example:

```
>>> odd = (1, 3, 5)
>>> even = (2, 4, 6)
>>> for i, j in zip(odd, even):
...     print(i + j)
...
3
7
11
```

By default, `zip()` will stop when any of the iterables is exhausted. You can set the `strict` keyword argument to `True` to raise an exception instead. See itertools.zip_longest() and stackoverflow: Zipped Python generators with 2nd one being shorter for alternatives.

```
>>> s1 = 'apple'
>>> s2 = 'fig'
>>> for c1, c2 in zip(s1, s2, strict=True):
...     print(f'{c1}:{c2}')
...
a:f
p:i
p:g
Traceback (most recent call last):
  File "<python-input-2>", line 1, in <module>
    for c1, c2 in zip(s1, s2, strict=True):
                  ~~~^^^^^^^^^^^^^^^^^^^^^
ValueError: zip() argument 2 is shorter than argument 1
```

As an **exercise**, write a function that returns the sum of product of corresponding elements of two sequences. For example, the result should be `44` for `(1, 3, 5)` and `(2, 4, 6)` .

Tuple methods

While this book won't discuss Object-Oriented Programming in any detail, you'll still see plenty examples for *using* them. You've already seen a few examples with modules. See Practical Python Programming and Fluent Python if you want to learn about Python OOP in depth. See also docs.python: Data model.

Data types in Python are all internally implemented as **classes**. You can use the dir() built-in function to get a list of valid attributes for an object.

```
# you can also use tuple objects such as 'odd' and 'even' declared earlier
>>> dir(tuple)
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count', 'index']
```

```
>>> even = (2, 4, 6)
# same as: len(even)
>>> even.__len__()
3
```

The non-dunder names (last two items) in the above listing will be discussed in this section. But first, a refresher on the `in` membership operator is shown below.

```
>>> num = 5
>>> num in (10, 21, 33)
False

>>> num = 21
>>> num in (10, 21, 33)
True
```

The `count()` method returns the number of times a value is present in the `tuple` object.

```
>>> nums = (1, 4, 6, 22, 3, 5, 2, 1, 51, 3, 1)
>>> nums.count(3)
2
>>> nums.count(31)
0
```

The `index()` method will give the index of the first occurrence of a value. It will raise `ValueError` if the value isn't present, which you can avoid by using the `in` operator first. Or, you can use the `try-except` statement to handle the exception as needed.

```
>>> nums = (1, 4, 6, 22, 3, 5, 2, 1, 51, 3, 1)

>>> nums.index(3)
4

>>> n = 31
>>> nums.index(n)
Traceback (most recent call last):
  File "<python-input-3>", line 1, in <module>
    nums.index(n)
    ~~~~~~~~~~^^^
ValueError: tuple.index(x): x not in tuple
>>> if n in nums:
...     print(nums.index(n))
... else:
...     print(f'{n} not present in "nums" tuple')
...
31 not present in "nums" tuple
```

The `list` and `str` sequence types have many more methods and they will be discussed separately in later chapters.

Specialized container datatypes

- docs.python: collections — alternatives to Python’s general purpose built-in containers, `dict` , `list` , `set` , and `tuple`
- docs.python: array — compactly represent an array of basic values: characters, integers, floating point numbers
- boltons — pure-Python utilities which extend the Python standard library

List

List is a container data type, similar to `tuple`, with lots of added functionality. You can modify a `list` container and thus it is a **mutable** data type. Lists are typically used to store and manipulate ordered collection of values.

Initialization and Slicing

Lists are declared as a comma separated values within square `[]` brackets. Unlike `tuple` there's no ambiguity in using `[]` characters, so there's no special requirement of trailing comma to declare a single element container. You can use a trailing comma if you wish, which is helpful to easily change a `list` declared across multiple lines.

```
# 1D example
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels[0]
'a'
# same as vowels[4] since len(vowels) - 1 = 4
>>> vowels[-1]
'u'

# 2D example
>>> student = ['learnbyexample', 2024, ['Linux', 'Vim', 'Python']]
>>> student[1]
2024
>>> student[2]
['Linux', 'Vim', 'Python']
>>> student[2][-1]
'Python'
```

Since `list` is a mutable data type, you can modify the container after initialization. You can either change a single element or use slicing notation to modify multiple elements.

```
>>> nums = [1, 4, 6, 22, 3, 5]

>>> nums[0] = 100
>>> nums
[100, 4, 6, 22, 3, 5]

>>> nums[-3:] = [-1, -2, -3]
>>> nums
[100, 4, 6, -1, -2, -3]

# list will automatically shrink or expand as needed
>>> nums[1:4] = [2000]
>>> nums
[100, 2000, -2, -3]
>>> nums[1:2] = [3.14, 4.13, 6.78]
>>> nums
[100, 3.14, 4.13, 6.78, -2, -3]
```

List methods and operations

This section will discuss some of the `list` methods and operations. See docs.python: list methods for documentation. As mentioned earlier, you can use `dir(list)` to view the available methods.

Use the `append()` method to add a single element to the end of a `list` object. If you need to append multiple items, you can pass an iterable to the `extend()` method. As an **exercise**, check what happens if you pass an iterable to the `append()` method and a non-iterable value to the `extend()` method. What happens if you pass multiple values to both these methods?

```
>>> books = []
>>> books.append('Cradle')
>>> books.append('Mistborn')
>>> books
['Cradle', 'Mistborn']

>>> items = [3, 'apple', 100.23]
>>> items.extend([4, 'mango'])
>>> items
[3, 'apple', 100.23, 4, 'mango']
>>> items.extend((-1, -2))
>>> items.extend(range(3))
>>> items.extend('hi')
>>> items
[3, 'apple', 100.23, 4, 'mango', -1, -2, 0, 1, 2, 'h', 'i']
```

The `count()` method will give the number of times a value is present.

```
>>> nums = [1, 4, 6, 22, 3, 5, 2, 1, 51, 3, 1]
>>> nums.count(3)
2
>>> nums.count(31)
0
```

The `index()` method will give the index of the first occurrence of a value. As seen with `tuple`, this method will raise `ValueError` if the value isn't present.

```
>>> nums = [1, 4, 6, 22, 3, 5, 2, 1, 51, 3, 1]
>>> nums.index(3)
4
```

The `pop()` method removes the last element of a `list` by default. You can pass an index to delete that specific item and the list will be automatically re-arranged. Return value is the element being deleted.

```
>>> primes = [2, 3, 5, 7, 11]
>>> last = primes.pop()
>>> last
11
>>> primes
[2, 3, 5, 7]
>>> primes.pop(2)
```

```
5
>>> primes
[2, 3, 7]

>>> student = ['learnbyexample', 2024, ['Linux', 'Vim', 'Python']]
>>> student.pop(1)
2024
>>> student[-1].pop(1)
'Vim'
>>> student
['learnbyexample', ['Linux', 'Python']]
>>> student.pop()
['Linux', 'Python']
>>> student
['learnbyexample']
```

The `del` statement is helpful to remove multiple elements using the slicing notation. Unlike the `pop()` method, there is no return value.

```
>>> nums = [1.2, -0.2, 0, 567, 9082, 23]
>>> del nums[0]
>>> nums
[-0.2, 0, 567, 9082, 23]
>>> del nums[2:4]
>>> nums
[-0.2, 0, 23]

>>> nums_2d = [[1, 3, 2, 10], [1.2, -0.2, 0, 2], [100, 200]]
>>> del nums_2d[0][1:3]
>>> del nums_2d[1]
>>> nums_2d
[[1, 10], [100, 200]]
```

The `pop()` method deletes an element based on its index. Use the `remove()` method to delete an element based on its value. You'll get `ValueError` if the value isn't found.

```
>>> even_numbers = [2, 4, 6, 8, 10]
>>> even_numbers.remove(8)
>>> even_numbers
[2, 4, 6, 10]
```

The `clear()` method removes all the elements. You might wonder why not just assign an empty `list`? If you have observed closely, all of the methods seen so far modified the `list` object in-place. This is useful if you are passing a `list` object to a function and expect the function to modify the object itself instead of returning a new object. See the Mutability chapter for more details.

```
>>> nums = [1.2, -0.2, 0, 2, 4, 23]
>>> nums.clear()
>>> nums
[]
```

You've already seen how to add elements at the end of a `list` using the `append()` and `extend()` methods. The `insert()` method helps you place an object at the given index. As an **exercise**, check what happens if you pass a `list` value. What happens if you pass more than one object?

```
>>> books = ['Sourdough', 'Sherlock Holmes', 'To Kill a Mocking Bird']
>>> books.insert(2, 'The Martian')
>>> books
['Sourdough', 'Sherlock Holmes', 'The Martian', 'To Kill a Mocking Bird']
```

The `reverse()` method reverses a `list` object in-place. Use slicing notation if you want a new object.

```
>>> primes = [2, 3, 5, 7, 11]
>>> primes.reverse()
>>> primes
[11, 7, 5, 3, 2]

>>> primes[::-1]
[2, 3, 5, 7, 11]
>>> primes
[11, 7, 5, 3, 2]
```

Here are some examples with comparison operators. Quoting from documentation:

> For two collections to compare equal, they must be of the same type, have the same length, and each pair of corresponding elements must compare equal (for example, `[1,2] == (1,2)` is false because the type is not the same).
> Collections that support order comparison are ordered the same as their first unequal elements (for example, `[1,2,x] <= [1,2,y]` has the same value as `x <= y`). If a corresponding element does not exist, the shorter collection is ordered first (for example, `[1,2] < [1,2,3]` is true).

```
>>> primes = [2, 3, 5, 7, 11]
>>> nums = [2, 3, 5, 11, 7]
>>> primes == nums
False
>>> primes == [2, 3, 5, 7, 11]
True

>>> [1, 1000] < [2, 3]
True
>>> [1000, 2] < [1, 2, 3]
False

>>> ['a', 'z'] > ['a', 'x']
True
>>> [1, 2, 3] > [10, 2]
False
>>> [1, 2, 3] > [1, 2]
True
```

Sorting and company

The `sort()` method will order the `list` object in-place. The sorted() built-in function provides the same functionality for iterable types and returns an ordered `list` .

```
>>> nums = [1, 5.3, 321, 0, 1, 2]

# ascending order
>>> nums.sort()
>>> nums
[0, 1, 1, 2, 5.3, 321]

# descending order
>>> nums.sort(reverse=True)
>>> nums
[321, 5.3, 2, 1, 1, 0]

>>> sorted('fuliginous')
['f', 'g', 'i', 'i', 'l', 'n', 'o', 's', 'u', 'u']
```

The `key` argument accepts the name of a function (i.e. the object) for custom sorting. If two elements are deemed equal based on the result of the function, the original order will be maintained (known as **stable sorting**). Here are some examples:

```
# based on the absolute value of an element
# note that the input order is maintained for all three values of "4"
>>> sorted([-1, -4, 309, 4.0, 34, 0.2, 4], key=abs)
[0.2, -1, -4, 4.0, 4, 34, 309]

# based on the length of an element
>>> words = ('morello', 'irk', 'fuliginous', 'crusado', 'seam')
>>> sorted(words, key=len, reverse=True)
['fuliginous', 'morello', 'crusado', 'seam', 'irk']
```

If the custom user-defined function required is just a single expression, you can create anonymous functions with lambda expressions instead of a full-fledged function. As an **exercise**, read docs.python HOWTOs: Sorting and implement the below examples using the `operator` module instead of `lambda` expressions.

```
# based on the second element of each item
>>> items = [('bus', 10), ('car', 20), ('jeep', 3), ('cycle', 5)]
>>> sorted(items, key=lambda e: e[1], reverse=True)
[('car', 20), ('bus', 10), ('cycle', 5), ('jeep', 3)]

# based on the number of words, assuming space as the word separator
>>> dishes = ('Poha', 'Aloo tikki', 'Baati', 'Khichdi', 'Makki roti')
>>> sorted(dishes, key=lambda s: s.count(' '), reverse=True)
['Aloo tikki', 'Makki roti', 'Poha', 'Baati', 'Khichdi']
```

You can use sequence types like `list` or `tuple` to specify multiple sorting conditions. Make sure to read the sequence comparison examples from the previous section before trying to understand the following examples.

```
>>> dishes = ('Poha', 'Aloo tikki', 'Baati', 'Khichdi', 'Makki roti')

# word-count and dish-names, both descending order
>>> sorted(dishes, key=lambda s: (s.count(' '), s), reverse=True)
['Makki roti', 'Aloo tikki', 'Poha', 'Khichdi', 'Baati']

# word-count descending order, dish-names ascending order
# the main trick is to negate the numerical value
>>> sorted(dishes, key=lambda s: (-s.count(' '), s))
['Aloo tikki', 'Makki roti', 'Baati', 'Khichdi', 'Poha']
```

As an **exercise**, given `nums = [1, 4, 5, 2, 51, 3, 6, 22]`, determine and implement the sorting condition based on the required output shown below:

- `[4, 2, 6, 22, 1, 5, 51, 3]`
- `[2, 4, 6, 22, 1, 3, 5, 51]`
- `[22, 6, 4, 2, 51, 5, 3, 1]`

Here are some examples with the `min()` and `max()` functions.

```
>>> nums = [321, 0.5, 899.232, 5.3, 2, 1, -1]
>>> min(nums)
-1
>>> max(nums)
899.232
>>> min(nums, key=abs)
0.5
```

Random items

You have already seen a few examples for the `random` module in earlier chapters. This section will show examples for methods that act on sequence data types.

First up, getting a random element from a non-empty sequence using the `choice()` method.

```
>>> import random

>>> random.choice([4, 5, 2, 76])
76
>>> random.choice('hello')
'e'
```

The `shuffle()` method randomizes the elements of a `list` in-place.

```
>>> items = ['car', 20, 3, 'jeep', -3.14, 'hi']

>>> random.shuffle(items)
>>> items
['car', 3, -3.14, 'jeep', 'hi', 20]
```

Use the `sample()` method to get a `list` of specified number of random elements. As an **exercise**, see what happens if you pass a slice size greater than the number of elements present in the input sequence.

```
>>> random.sample((4, 5, 2, 76), k=3)
[4, 76, 2]

>>> random.sample(range(1000), k=5)
[490, 26, 9, 745, 919]
```

Map, Filter and Reduce

Many operations on container objects can be defined in terms of these three concepts. For example, if you want to sum the square of all even numbers:

- separating out even numbers is **Filter** (i.e. only elements that satisfy a condition are retained)
- square of such numbers is **Map** (i.e. each element is transformed by a mapping function)
- final sum is **Reduce** (i.e. you get one value out of many)

One or more of these operations may be absent depending on the problem statement. A function for the first of these steps could look like:

```
>>> def get_evens(iterable):
...     op = []
...     for n in iterable:
...         if n % 2 == 0:
...             op.append(n)
...     return op
...
>>> get_evens([100, 53, 32, 0, 11, 5, 2])
[100, 32, 0, 2]
```

Function after the second step could be:

```
>>> def sqr_evens(iterable):
...     op = []
...     for n in iterable:
...         if n % 2 == 0:
...             op.append(n * n)
...     return op
...
>>> sqr_evens([100, 53, 32, 0, 11, 5, 2])
[10000, 1024, 0, 4]
```

And finally, the function after the third step could be:

```
>>> def sum_sqr_evens(iterable):
...     total = 0
...     for n in iterable:
...         if n % 2 == 0:
...             total += n * n
...     return total
...
>>> sum_sqr_evens([100, 53, 32, 0, 11, 5, 2])
11028
```

Python also provides map(), filter() and functools.reduce() for such problems. See the Comprehensions and Generator expressions chapter before deciding to use them.

Here are some examples with the sum(), all() and any() built-in reduce functions.

```
>>> sum([321, 0.5, 899.232, 5.3, 2, 1, -1])
1228.032

>>> conditions = [True, False, True]
# True only if all the elements are Truthy
>>> all(conditions)
False
# True if at least one element is Truthy
>>> any(conditions)
True
>>> conditions[1] = True
>>> all(conditions)
True

>>> nums = [321, 1, 1, 0, 5.3, 2]
>>> all(nums)
False
>>> any(nums)
True
```

Exercises

- Write a function that returns the product of a sequence of numbers. Empty sequence or sequence containing non-numerical values should raise `TypeError` .
  - `product([-4, 2.3e12, 77.23, 982, 0b101])` should give `-3.48863356e+18`
  - `product(range(2, 6))` should give `120`
  - `product(())` and `product(['a', 'b'])` should raise `TypeError`
- Write a function that removes dunder names from the `dir()` output.

  ```
  >>> remove_dunder(list)
  ['append', 'clear', 'copy', 'count', 'extend', 'index',
   'insert', 'pop', 'remove', 'reverse', 'sort']
  >>> remove_dunder(tuple)
  ['count', 'index']
  ```

Mutability

`int` , `float` , `str` and `tuple` are examples for immutable data types. On the other hand, types like `list` and `dict` are mutable. This chapter will discuss what happens when you pass a variable to a function or when you assign them to another value/variable.

id

The id() built-in function returns the *identity* (reference) of an object. Here are some examples to show what happens when you assign a variable to another value/variable.

```
>>> num1 = 5
>>> id(num1)
140204812958128
# here, num1 gets a new identity
>>> num1 = 10
>>> id(num1)
140204812958288

# num2 will have the same reference as num1
>>> num2 = num1
>>> id(num2)
140204812958288

# num2 gets a new reference, num1 won't be affected
>>> num2 = 4
>>> id(num2)
140204812958096
>>> num1
10
```

Pass by reference

Variables in Python store references to an object, not their values. When you pass a `list` object to a function, you are passing the reference to this object. Since `list` is mutable, any in-place changes made to this object within the function will also be reflected in the original variable that was passed to the function. Here's an example:

```
>>> def rotate(ip):
...     ip.insert(0, ip.pop())
...
>>> nums = [321, 1, 1, 0, 5.3, 2]
>>> rotate(nums)
>>> nums
[2, 321, 1, 1, 0, 5.3]
```

This is true even for slices of a sequence containing mutable objects. Also, as shown in the example below, `tuple` doesn't prevent mutable elements from being changed.

```
>>> nums_2d = ([1, 3, 2, 10], [1.2, -0.2, 0, 2], [100, 200])
>>> last_two = nums_2d[-2:]
>>> last_two[0][-1] = 'apple'
>>> last_two[1][-1] = 'ball'

>>> last_two
([1.2, -0.2, 0, 'apple'], [100, 'ball'])
>>> nums_2d
([1, 3, 2, 10], [1.2, -0.2, 0, 'apple'], [100, 'ball'])
```

As an **exercise**, use the `id()` function to verify that the identity of the last two elements of the `nums_2d` variable in the above example is the same as the identity of both the elements in the `last_two` variable.

Slicing notation shallow copy

If you wish to copy whole or part of a `list` object such that changing the copy version doesn't affect the original `list`, the solution will depend on the presence of mutable elements.

Here's an example where all the elements are immutable. In this case, using slice notation is safe for copying.

```
>>> items = [3, 'apple', 100.23, 'fig']
>>> items_copy = items[:]

>>> id(items)
140204765864256
>>> id(items_copy)
140204765771968

# the individual elements will still have the same reference
>>> id(items[0]) == id(items_copy[0])
True

>>> items_copy[0] += 1000
>>> items_copy
[1003, 'apple', 100.23, 'fig']
>>> items
[3, 'apple', 100.23, 'fig']
```

On the other hand, if the sequence has mutable objects, a shallow copy made using slicing notation won't stop the copy from modifying the original.

```
>>> nums_2d = [[1, 3, 2, 10], [1.2, -0.2, 0, 2], [100, 200]]
>>> nums_2d_copy = nums_2d[:]

>>> nums_2d_copy[0][0] = 'oops'

>>> nums_2d_copy
[['oops', 3, 2, 10], [1.2, -0.2, 0, 2], [100, 200]]
>>> nums_2d
[['oops', 3, 2, 10], [1.2, -0.2, 0, 2], [100, 200]]
```

copy.deepcopy

The copy built-in module has a `deepcopy()` method if you wish to recursively create new copies of all the elements of a mutable object.

```
>>> import copy

>>> nums_2d = [[1, 3, 2, 10], [1.2, -0.2, 0, 2], [100, 200]]
>>> nums_2d_deepcopy = copy.deepcopy(nums_2d)

>>> nums_2d_deepcopy[0][0] = 'yay'

>>> nums_2d_deepcopy
[['yay', 3, 2, 10], [1.2, -0.2, 0, 2], [100, 200]]
>>> nums_2d
[[1, 3, 2, 10], [1.2, -0.2, 0, 2], [100, 200]]
```

As an **exercise**, create a deepcopy of only the first two elements of the `nums_2d` object from the above example.

Dict

Dictionaries can be thought of as a collection of *key-value* pairs or a *named list of items*. It used to be unordered, but Python now ensures that the insertion order is maintained. See this tutorial for a more detailed discussion on `dict` usage.

Initialization and accessing elements

A `dict` data type is declared within `{}` characters and each item requires two values — an immutable data type for keys, followed by a `:` character and finally a value of any data type. The elements are separated by a comma character, just like the other container types.

To access an element, the syntax is `dict_variable[key]`. Retrieving an item takes a constant amount of time, irrespective of the size of the `dict` (see Hashtables for details). Dictionaries are mutable, so you can change an item's value, add items, remove items, etc.

```
>>> marks = {'Rahul': 86, 'Ravi': 92, 'Rohit': 75, 'Rajan': 79}

>>> marks['Rohit']
75
>>> marks['Rahul'] += 5
>>> marks['Ram'] = 67
>>> del marks['Rohit']
# note that the insertion order is maintained
>>> marks
{'Rahul': 91, 'Ravi': 92, 'Rajan': 79, 'Ram': 67}
```

Here's an example with `list` and `tuple` keys.

```
>>> list_key = {[1, 2]: 42}
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    list_key = {[1, 2]: 42}
               ^^^^^^^^^^^^
TypeError: unhashable type: 'list'

>>> items = {('car', 2): 'honda', ('car', 5): 'tesla', ('bike', 10): 'hero'}
>>> items[('bike', 10)]
'hero'
```

You can also use the dict() function for initialization in various ways. If all the keys are of `str` data type, you can use the same syntax as keyword arguments seen earlier with function definitions. You can also pass a container type having two values per element, such as a `list` of `tuples` as shown below.

```
>>> marks = dict(Rahul=86, Ravi=92, Rohit=75, Rajan=79)
>>> marks
{'Rahul': 86, 'Ravi': 92, 'Rohit': 75, 'Rajan': 79}

>>> items = [('jeep', 20), ('car', 3), ('cycle', 5)]
>>> dict(items)
{'jeep': 20, 'car': 3, 'cycle': 5}
```

Another way to initialize is to use the `fromkeys()` method that accepts an iterable and an optional value (default is `None`). The same value will be assigned to all the keys, so be careful if you want to use a mutable object, since the same reference will be used as well.

```
>>> colors = ('red', 'blue', 'green')
>>> dict.fromkeys(colors)
{'red': None, 'blue': None, 'green': None}
>>> dict.fromkeys(colors, 255)
{'red': 255, 'blue': 255, 'green': 255}
```

get and setdefault

If you try to access a `dict` key that doesn't exist, you'll get a `KeyError` exception. If you do not want an exception to occur, you can use the `get()` method. By default it'll return a `None` value for keys that do not exist, which you can change by providing a default value as the second argument.

```
>>> marks = dict(Rahul=86, Ravi=92, Rohit=75, Rajan=79)

>>> marks['Ron']
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    marks['Ron']
    ~~~~~^^^^^^^
KeyError: 'Ron'

>>> marks.get('Ravi')
92
>>> value = marks.get('Ron')
>>> print(value)
None
>>> marks.get('Ron', 0)
0
```

Here's a more practical example:

```
>>> vehicles = ['car', 'jeep', 'car', 'bike', 'bus', 'car', 'bike']
>>> hist = {}
>>> for v in vehicles:
...     hist[v] = hist.get(v, 0) + 1
...
>>> hist
{'car': 3, 'jeep': 1, 'bike': 2, 'bus': 1}
```

Using the `get()` method will not automatically add keys that do not exist yet to the `dict` object. You can use the `setdefault()` method, which behaves similarly to `get()` except that keys will get created if not found. See also docs.python: collections.defaultdict.

```
>>> marks = dict(Rahul=86, Ravi=92, Rohit=75, Rajan=79)

>>> marks.get('Ram', 40)
40
```

```
>>> marks
{'Rahul': 86, 'Ravi': 92, 'Rohit': 75, 'Rajan': 79}

>>> marks.setdefault('Ram', 40)
40
>>> marks
{'Rahul': 86, 'Ravi': 92, 'Rohit': 75, 'Rajan': 79, 'Ram': 40}
```

Iteration

The default `for` loop over a `dict` object will give you a key for each iteration.

```
>>> fruits = dict(banana=12, papaya=5, mango=10, fig=100)

>>> for k in fruits:
...     print(f'{k}:{fruits[k]}')
...
banana:12
papaya:5
mango:10
fig:100

# similarly, you'll get only the keys if you apply list(), tuple() or set()
>>> list(fruits)
['banana', 'papaya', 'mango', 'fig']
```

As an **exercise**,

- given `fruits` dictionary as defined in the above code snippet, what do you think will happen when you use `a, *b, c = fruits` ?
- given `nums = [1, 4, 6, 22, 3, 5, 4, 3, 6, 2, 1, 51, 3, 1]` , keep only the first occurrences of a value from this list without changing the order of elements. You can do it with the `dict` features presented so far. `[1, 4, 6, 22, 3, 5, 2, 51]` should be the output.

Dict methods and operations

The `in` operator checks if a key is present in the given dictionary. The `keys()` method returns all the keys and the `values()` method returns all the values. These methods return a custom set-like object, but with the insertion order maintained.

```
>>> marks = dict(Rahul=86, Ravi=92, Rohit=75, Rajan=79)

>>> 'Ravi' in marks
True
>>> 'Ram' in marks
False

>>> marks.keys()
dict_keys(['Rahul', 'Ravi', 'Rohit', 'Rajan'])
```

```
>>> marks.values()
dict_values([86, 92, 75, 79])
```

The `items()` method can be used to get a key-value `tuple` for each iteration.

```
>>> fruits = dict(banana=12, papaya=5, mango=10, fig=100)

# set-like object
>>> fruits.items()
dict_items([('banana', 12), ('papaya', 5), ('mango', 10), ('fig', 100)])

>>> for fruit, qty in fruits.items():
...     print(f'{fruit}\t: {qty}')
...
banana  : 12
papaya  : 5
mango   : 10
fig     : 100
```

The `del` statement example seen earlier removes the given key without returning the value associated with it. You can use the `pop()` method to get the value as well. The `popitem()` method removes the last added item and returns the key-value pair as a `tuple` .

```
>>> marks = dict(Rahul=86, Ravi=92, Rohit=75, Rajan=79)

>>> marks.pop('Ravi')
92
>>> marks
{'Rahul': 86, 'Rohit': 75, 'Rajan': 79}

>>> marks.popitem()
('Rajan', 79)
>>> marks
{'Rahul': 86, 'Rohit': 75}
```

The `update()` method allows you to add/update items from another dictionary or a container with key-value pair elements.

```
>>> marks = dict(Rahul=86, Ravi=92, Rohit=75, Rajan=79)
>>> marks.update(dict(Jo=89, Joe=75, Ravi=100))
# note that 'Ravi' has '100' as the updated value
>>> marks
{'Rahul': 86, 'Ravi': 100, 'Rohit': 75, 'Rajan': 79, 'Jo': 89, 'Joe': 75}

>>> fruits = dict(papaya=5, mango=10, fig=100)
>>> fruits.update([('tomato', 3), ('banana', 10)])
>>> fruits
{'papaya': 5, 'mango': 10, 'fig': 100, 'tomato': 3, 'banana': 10}
```

The `|` operator is similar to the `update()` method, except that you get a new `dict` object instead of in-place modification.

```
>>> d1 = {'banana': 12, 'papaya': 5, 'mango': 20}
>>> d2 = {'mango': 10, 'fig': 100}

# before the introduction of the | operator,
# you had to use unpacking, i.e. {**d1, **d2}
>>> d1 | d2
{'banana': 12, 'papaya': 5, 'mango': 10, 'fig': 100}
```

Arbitrary keyword arguments

To accept an arbitrary number of keyword arguments, use `**var_name` in the function definition. This has to be declared the last, after all the other types of arguments. Idiomatically, `**kwargs` is used as the variable name. See stackoverflow: Decorators demystified for a practical example.

```
>>> def many(**kwargs):
...     print(f'{kwargs = }')
...
>>> many()
kwargs = {}
>>> many(num=5)
kwargs = {'num': 5}
>>> many(car=5, jeep=25)
kwargs = {'car': 5, 'jeep': 25}
```

Turning it around, when you have a function defined with keyword arguments, you can unpack a dictionary while calling the function.

```
>>> def greeting(phrase='hello', style='='):
...     print(f'{phrase:{style}^{len(phrase)+6}}')
...
>>> greeting()
===hello===
>>> d = {'style': '-', 'phrase': 'have a nice day'}
>>> greeting(**d)
---have a nice day---
```

Set

`set` is a mutable, unordered collection of hashable objects. `frozenset` is similar to `set`, but immutable. See docs.python: set, frozenset for documentation.

Initialization

Sets are declared as a collection of objects separated by a comma within `{}` characters. The set() function can be used to initialize an empty `set` and to convert iterables.

```
>>> empty_set = set()
>>> empty_set
set()

>>> nums = {-0.1, 3, 2, -5, 7, 1, 6.3, 5}
# note that the order is not the same as declaration
>>> nums
{-0.1, 1, 2, 3, 5, 6.3, 7, -5}

# sets can only contain distinct elements
>>> set([3, 2, 11, 3, 5, 13, 2])
{2, 3, 5, 11, 13}
>>> set('initialize')
{'a', 'n', 't', 'l', 'e', 'i', 'z'}
```

`set` doesn't allow mutable objects as elements.

```
>>> {1, 3, [1, 2], 4}
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    {1, 3, [1, 2], 4}
TypeError: unhashable type: 'list'

>>> {1, 3, (1, 2), 4}
{3, 1, (1, 2), 4}
```

Set methods and operations

The `in` operator checks if a value is present in the given `set`. Since `set` uses a hashtable (similar to `dict` keys), the lookup time is constant and much faster than ordered collections like `list` or `tuple` for large containers.

```
>>> colors = {'red', 'blue', 'green'}
>>> 'blue' in colors
True
>>> 'orange' in colors
False
```

Here are some examples for `set` operations like union, intersection, etc. You can either use methods or operators, both will give you a new `set` object instead of in-place modification. The difference is that the `set` methods can accept any iterable, whereas the operators can work only with `set` or set-like objects.

```
>>> color_1 = {'teal', 'light blue', 'green', 'yellow'}
>>> color_2 = {'light blue', 'black', 'dark green', 'yellow'}

# union of two sets: color_1 | color_2
>>> color_1.union(color_2)
{'light blue', 'green', 'dark green', 'black', 'teal', 'yellow'}

# common items: color_1 & color_2
>>> color_1.intersection(color_2)
{'light blue', 'yellow'}

# items from color_1 not present in color_2: color_1 - color_2
>>> color_1.difference(color_2)
{'teal', 'green'}
# items from color_2 not present in color_1: color_2 - color_1
>>> color_2.difference(color_1)
{'dark green', 'black'}

# items present in one of the sets, but not both
# i.e. union of previous two operations: color_1 ^ color_2
>>> color_1.symmetric_difference(color_2)
{'green', 'dark green', 'black', 'teal'}
```

As mentioned in the Dict chapter, methods like `keys()` , `values()` and `items()` return a set-like object. You can apply `set` operators on them.

```
>>> marks_1 = dict(Rahul=86, Ravi=92, Rohit=75)
>>> marks_2 = dict(Jo=89, Rohit=78, Joe=75, Ravi=100)

>>> marks_1.keys() & marks_2.keys()
{'Ravi', 'Rohit'}
>>> marks_1.keys() - marks_2.keys()
{'Rahul'}
```

Methods like `add()` , `update()` , `symmetric_difference_update()` , `intersection_update()` and `difference_update()` will do the modifications in-place.

```
>>> color_1 = {'teal', 'light blue', 'green', 'yellow'}
>>> color_2 = {'light blue', 'black', 'dark green', 'yellow'}

# union
>>> color_1.update(color_2)
>>> color_1
{'light blue', 'green', 'dark green', 'black', 'teal', 'yellow'}

# adding a single value
>>> color_2.add('orange')
>>> color_2
{'black', 'yellow', 'dark green', 'light blue', 'orange'}
```

The `pop()` method will return a random element being removed. Use the `remove()` method if you want to delete an element based on its value. The `discard()` method is similar to `remove()`, but it will not generate an error if the element doesn't exist. The `clear()` method will delete all the elements.

```
>>> colors = {'red', 'blue', 'green'}

>>> colors.pop()
'blue'
>>> colors
{'green', 'red'}

# you'll get KeyError if you use the 'remove()' method here
>>> colors.discard('black')

>>> colors.clear()
>>> colors
set()
```

Here are some examples for comparison operations.

```
>>> names_1 = {'Ravi', 'Rohit'}
>>> names_2 = {'Ravi', 'Ram', 'Rohit', 'Raj'}

>>> names_1 == names_2
False

# same as: names_1 <= names_2
>>> names_1.issubset(names_2)
True

# same as: names_2 >= names_1
>>> names_2.issuperset(names_1)
True

# disjoint checks if there are no common elements
# same as: not names_1 & names_2
>>> names_1.isdisjoint(names_2)
False
>>> names_1.isdisjoint({'Jo', 'Joe'})
True
```

Exercises

- Write a function that checks whether an iterable has duplicate values or not.

  ```
  >>> has_duplicates('pip')
  True
  >>> has_duplicates((3, 2))
  False
  ```

- What does the above function return for `has_duplicates([3, 2, 3.0])` ?

Text processing

This chapter will primarily focus on `str` methods to solve a wide variety of text processing tasks. You'll also see a few examples using the `string` and `re` modules.

join

The `join()` method is similar to what the `print()` function does with the `sep` option, except that you get a `str` object as the result. The iterable you pass to `join()` must only have string elements.

```
>>> print(1, 2)
1 2
>>> ' '.join((1, 2))
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    ' '.join((1, 2))
    ~~~~~~~~^^^^^^^^
TypeError: sequence item 0: expected str instance, int found
>>> ' '.join(('1', '2'))
'1 2'

>>> c = ' :: '
>>> c.join(['This', 'is', 'a', 'sample', 'string'])
'This :: is :: a :: sample :: string'
```

As an **exercise**, check what happens if you pass multiple string values separated by comma to `join()` instead of an iterable.

The `print()` method uses an object's `__str__()` method to get its string representation. The `__repr__()` method is used as a fallback.

Transliteration

The `translate()` method accepts a table of codepoints (numerical value of a character) mapped to another character/codepoint or `None` (if the character has to be deleted). You can use the ord() built-in function to get the codepoint of characters. Or, you can use the `str.maketrans()` method to generate the mapping for you.

```
>>> ord('a')
97
>>> ord('A')
65

>>> str.maketrans('aeiou', 'AEIOU')
{97: 65, 101: 69, 105: 73, 111: 79, 117: 85}

>>> greeting = 'have a nice day'
```

```
>>> greeting.translate(str.maketrans('aeiou', 'AEIOU'))
'hAvE A nIcE dAy'
```

The string module has a collection of constants that are often useful in text processing. Here's an example of deleting punctuation characters.

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'

>>> para = '"Hi", there! How *are* you? All fine here.'
>>> para.translate(str.maketrans('', '', string.punctuation))
'Hi there How are you All fine here'

>>> chars_to_delete = ''.join(set(string.punctuation) - set('.!?'))
>>> para.translate(str.maketrans('', '', chars_to_delete))
'Hi there! How are you? All fine here.'
```

As an **exercise**, read the documentation for features covered in this section. See also stackoverflow: character translation examples.

Removing leading and trailing characters

The `strip()` method removes consecutive characters from the start/end of the given string. By default this method removes whitespace characters, which you can change by passing a `str` argument. You can use the `lstrip()` and `rstrip()` methods to work only on the leading and trailing characters respectively.

```
>>> greeting = '  \t\r\n have    a  nice \t day  \f\v\r\t\n '
>>> greeting.strip()
'have    a  nice \t day'
>>> greeting.lstrip()
'have    a  nice \t day  \x0c\x0b\r\t\n '
>>> greeting.rstrip()
'  \t\r\n have    a  nice \t day'

>>> '"Hi". How are you!?'.strip(string.punctuation)
'Hi". How are you'
```

The `removeprefix()` and `removesuffix()` methods will delete a substring from the start/end of the input string.

```
>>> 'spare'.removeprefix('sp')
'are'
>>> 'free'.removesuffix('e')
'fre'
# difference between remove and strip
>>> 'cared'.removesuffix('de')
'cared'
# strip uses given argument as a set of characters to be removed in any order
>>> 'cared'.rstrip('de')
'car'
```

Dealing with case

Here are five different methods for changing the case of characters. Word level transformation is determined by consecutive occurrences of alphabets, not limited to separation by whitespace characters.

```
>>> sentence = 'thIs iS a saMple StrIng'

>>> sentence.capitalize()
'This is a sample string'

>>> sentence.title()
'This Is A Sample String'

>>> sentence.lower()
'this is a sample string'

>>> sentence.upper()
'THIS IS A SAMPLE STRING'

>>> sentence.swapcase()
'THiS Is A SAmPLE sTRiNG'
```

The `string.capwords()` method is similar to `title()` but also allows a specific word separator (whose default is whitespace).

```
>>> phrase = 'this-IS-a:colon:separated,PHRASE'

>>> phrase.title()
'This-Is-A:Colon:Separated,Phrase'
>>> string.capwords(phrase, ':')
'This-is-a:Colon:Separated,phrase'
```

is methods

The `islower()` , `isupper()` and `istitle()` methods check if the given string conforms to the specific case pattern. Characters other than alphabets do not influence the result, but at least one alphabet needs to be present for a `True` output.

```
>>> 'αλεπού'.islower()
True

>>> '123'.isupper()
False
>>> 'ABC123'.isupper()
True

>>> 'Today is Sunny'.istitle()
False
```

Here are some examples with the `isnumeric()` and `isascii()` methods. As an **exercise**, read the documentation for the rest of the **is** methods.

```
# checks if the string has numeric characters only (at least one)
>>> '153'.isnumeric()
True
>>> ''.isnumeric()
False
>>> '1.2'.isnumeric()
False
>>> '-1'.isnumeric()
False

# False if any character codepoint is outside the range 0x00 to 0x7F
>>> '123—456'.isascii()
False
>>> 'happy learning!'.isascii()
True
```

Substring and count

The `in` operator checks if the LHS string is a substring of the RHS string.

```
>>> sentence = 'This is a sample string'

>>> 'is a' in sentence
True
>>> 'this' in sentence
False
>>> 'this' in sentence.lower()
True
>>> 'test' not in sentence
True
```

The `count()` method gives the number of times the given substring is present (non-overlapping).

```
>>> sentence = 'This is a sample string'
>>> sentence.count('is')
2
>>> sentence.count('w')
0

>>> word = 'phototonic'
>>> word.count('oto')
1
```

Match at the start and end of strings

The `startswith()` and `endswith()` methods check for the presence of substrings only at the start and end of an input string.

```
>>> sentence = 'This is a sample string'
>>> sentence.startswith('This')
True
>>> sentence.startswith('is')
False

>>> sentence.endswith('ing')
True
>>> sentence.endswith('ly')
False
```

If you need to check for multiple conditions, pass a `tuple` argument.

```
>>> words = ['refuse', 'impossible', 'present', 'read']
>>> prefix = ('im', 're')
>>> for w in words:
...     if w.startswith(prefix):
...         print(w)
...
refuse
impossible
read
```

split

The `split()` method splits a string based on the given substring and returns a `list`. By default, whitespace characters are used for splitting. You can also control the number of splits.

```
>>> greeting = '  \t\r\n have    a  nice \t day  \f\v\r\t\n '
# note that the leading/trailing whitespaces do not create empty elements
>>> greeting.split()
['have', 'a', 'nice', 'day']

# note that the empty elements are preserved here
>>> ':car::jeep::'.split(':')
['', 'car', '', 'jeep', '', '']

>>> 'apple<=>grape<=>mango<=>fig'.split('<=>', maxsplit=1)
['apple', 'grape<=>mango<=>fig']
```

As an **exercise**, read the documentation for the `rsplit()`, `partition()` and `rpartition()` methods.

replace

Use the `replace()` method for substitution operations. An optional `count` keyword argument allows you to specify the number of replacements to be made.

```
>>> phrase = '2 be or not 2 be'

>>> phrase.replace('2', 'to')
'to be or not to be'
```

```
>>> phrase.replace('2', 'to', count=1)
'to be or not 2 be'

# recall that string is immutable, you'll need to re-assign if needed
>>> phrase
'2 be or not 2 be'
>>> phrase = phrase.replace('2', 'to')
>>> phrase
'to be or not to be'
```

re module

Regular Expressions is a versatile tool for text processing. Here are some common use cases:

- Sanitizing a string to ensure that it satisfies a known set of rules. For example, to check if a given string matches password rules.
- Filtering or extracting portions on an abstract level like alphabets, digits, punctuation and so on.
- Qualified string replacement. For example, at the start or the end of a string, only whole words, based on surrounding text, etc.

You can use the built-in `re` module to perform such tasks. Here are some examples:

```
>>> import re

# extract non-colon character sequences
>>> ip = ':car::jeep::'
# using the 'split' method will result in possible empty elements
>>> ip.split(':')
['', 'car', '', 'jeep', '', '']
# with regular expressions, you can choose to match only the non-empty portions
# [^:] is a character class to match non : characters
# + is a quantifier that matches the preceding element one or more times
>>> re.findall(r'[^:]+', ip)
['car', 'jeep']

# replace only whole words 'par' OR 'hand' with 'X'
# \b is an anchor to restrict the matching to the start/end of words
# () has many uses, helps to group common elements here
# similar to 'a(b+c)d = abd+acd' in maths, you get 'a(b|c)d = abd|acd'
# | is similar to the 'or' operator
>>> ip = 'par spare part hand handy unhanded'
>>> re.sub(r'\b(par|hand)\b', 'X', ip)
'X spare part X handy unhanded'
```

See my book Understanding Python re(gex)? for a detailed guide on regular expressions. You'll also get to learn the third-party `regex` module.

Exercises

- Write a function that checks if two strings are anagrams irrespective of case. Assume that the input is made up of alphabets only.

```
>>> anagram('god', 'Dog')
True
>>> anagram('beat', 'table')
False
>>> anagram('Beat', 'abet')
True
```

- Read the documentation and implement these formatting examples with the equivalent `str` methods.

```
>>> fruit = 'apple'

>>> f'{fruit:=>10}'
'=====apple'
>>> f'{fruit:=<10}'
'apple====='
>>> f'{fruit:=^10}'
'==apple==='

>>> f'{fruit:^10}'
'  apple   '
```

- Write a function that returns a `list` of words present in the input string.

```
>>> words('"Hi", there! How *are* you? All fine here.')
['Hi', 'there', 'How', 'are', 'you', 'All', 'fine', 'here']
>>> words('This-Is-A:Colon:Separated,Phrase')
['This', 'Is', 'A', 'Colon', 'Separated', 'Phrase']
```

Comprehensions and Generator expressions

This chapter will show how to use comprehensions and generator expressions for **map**, **filter** and **reduce** operations. You'll also learn about **iterators** and the `yield` statement.

Comprehensions

As mentioned earlier, Python provides the map() and filter() built-in functions. Comprehensions provide a terser and a (usually) faster way to implement them. However, the syntax can take a while to understand and get comfortable with.

The minimal requirement for a comprehension is a mapping expression (which could include a function call) and a loop. Here's an example:

```
>>> nums = (321, 1, 1, 0, 5.3, 2)

# manual implementation
>>> sqr_nums = []
>>> for n in nums:
...     sqr_nums.append(n * n)
...
>>> sqr_nums
[103041, 1, 1, 0, 28.09, 4]

# list comprehension
>>> [n * n for n in nums]
[103041, 1, 1, 0, 28.09, 4]
```

The general form of the above `list` comprehension is `[expr loop]`. Comparing with the manual implementation, the difference is that `append()` is automatically performed, which is where most of the performance benefit comes from. Note that the `list` comprehension is defined based on the output being a `list`, input to the `for` loop can be any iterable (like `tuple` in the above example).

Here's an example with a filtering operation. Instead of the following implementations:

```
# manual implementation
def remove_dunder(obj):
    names = []
    for n in dir(obj):
        if '__' not in n:
            names.append(n)
    return names

# using the 'filter' function
def remove_dunder(obj):
    return list(filter(lambda n: '__' not in n, dir(obj)))
```

You can use comprehension syntax like this:

```
>>> def remove_dunder(obj):
...     return [n for n in dir(obj) if '__' not in n]
...
```

```
>>> remove_dunder(dict)
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop',
 'popitem', 'setdefault', 'update', 'values']
```

The general form of the above comprehension is `[expr loop condition]`. If you can write the manual implementation, it is easy to derive the comprehension version. Put the expression (the argument passed to the `append()` method) first, and then put the loops and conditions in the same order as the manual implementation. With practice, you'll be able to read and write the comprehension versions naturally.

Here's an example with the `zip()` function:

```
>>> p = [1, 3, 5]
>>> q = [3, 214, 53]
>>> [i + j for i, j in zip(p, q)]
[4, 217, 58]
>>> [i * j for i, j in zip(p, q)]
[3, 642, 265]
```

And here's a nested loop example:

```
>>> names = ['Jo', 'Joe', 'Jon']
>>> pairs = []
>>> for i, n1 in enumerate(names):
...     for n2 in names[i+1:]:
...         pairs.append((n1, n2))
...
>>> pairs
[('Jo', 'Joe'), ('Jo', 'Jon'), ('Joe', 'Jon')]
# note that the loop order is same as the manual implementation
>>> [(n1, n2) for i, n1 in enumerate(names) for n2 in names[i+1:]]
[('Jo', 'Joe'), ('Jo', 'Jon'), ('Joe', 'Jon')]
```

Similarly, you can build `dict` and `set` comprehensions by using `{}` instead of the `[]` characters. Comprehension syntax inside `()` characters becomes a generator expression (discussed later in this chapter), so you'll need to use `tuple()` for `tuple` comprehension. You can use `list()`, `dict()` and `set()` instead of `[]` and `{}` respectively as well.

```
# filter by value comparison
>>> marks = dict(Rahul=68, Ravi=92, Rohit=75, Rajan=85, Ram=80)
>>> {k: v for k, v in marks.items() if v >= 80}
{'Ravi': 92, 'Rajan': 85, 'Ram': 80}

# filter by substring comparison
>>> colors = {'teal', 'blue', 'green', 'yellow', 'red', 'orange'}
>>> {c for c in colors if 'o' in c}
{'yellow', 'orange'}

# filter by the length of elements
>>> dishes = ('Poha', 'Aloo tikki', 'Baati', 'Khichdi', 'Makki roti')
>>> tuple(d for d in dishes if len(d) < 6)
('Poha', 'Baati')
```

If you are still confused with comprehension syntax, see:

- List comprehensions explained visually
- Comprehensions in Python the Jedi way
- calmcode.io: video on comprehensions

Iterator

Partial quote from docs.python glossary: **iterator**:

> An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead.

The `filter()` example in the previous section required further processing, such as passing to the `list()` function to get the output as a `list` object. This is because the `filter()` function returns an object that behaves like an **iterator**. You can pass iterators anywhere iterables are allowed, such as the `for` loop. Here's an example:

```
>>> filter_obj = filter(lambda n: '__' not in n, dir(tuple))
>>> filter_obj
<filter object at 0x7fd910e2de80>
>>> for x in filter_obj:
...     print(x)
...
count
index
```

One of the differences between an iterable and an iterator is that you can iterate over iterables any number of times (quite the tongue twister, if I may say so myself). Whereas, the `next()` function can be used on an iterator, but not iterables. Once you have exhausted an iterator, any attempt to get another item (such as `next()` or `for` loop) will result in a `StopIteration` exception. Iterators are lazy and memory efficient since the results are evaluated only when needed, instead of lying around in a container.

```
>>> names = filter(lambda n: '__' not in n, dir(tuple))
>>> next(names)
'count'
>>> next(names)
'index'
>>> next(names)
Traceback (most recent call last):
  File "<python-input-3>", line 1, in <module>
    next(names)
    ~~~~^^^^^^^
StopIteration
```

You can convert an iterable to an iterator using the iter() built-in function.

```
>>> nums = [321, 1, 1, 0, 5.3, 2]
>>> iter(nums)
```

```
<list_iterator object at 0x7fd90e7f8ee0>
```

Here's a practical example to get a random item from a `list` without repetition:

```
>>> import random
>>> names = ['Jo', 'Ravi', 'Joe', 'Raj', 'Jon']
>>> random.shuffle(names)
>>> random_name = iter(names)
>>> next(random_name)
'Jon'
>>> next(random_name)
'Ravi'
```

yield

In this section, you'll see an example with the `yield` statement to create an iterator known as **generators**. Quoting from docs.python: Generators:

> Each time `next()` is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed).

Here's a fibonacci generator:

```
>>> def fibonacci(n):
...     a, b = 0, 1
...     for _ in range(n):
...         yield a
...         a, b = b, a + b
...
>>> fibonacci(5)
<generator object fibonacci at 0x7fd90e7b22e0>
>>> list(fibonacci(10))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

For a more detailed discussion and related features, see:

- stackoverflow: What does the yield keyword do?
- Yield and Generators Explained
- docs.python: itertools

Generator expressions

Using comprehension syntax inside `()` characters creates an iterator, known as **generator expressions**. Compared to comprehensions, a generator expression is memory efficient and faster whenever you need a single use iterable. If you use comprehension, you'll be wasting memory to save the values in a container, only to be discarded once they are processed. For example, a **reduce** operation such as the `sum()` function as shown below.

```
>>> nums = [100, 53, 32, 0, 11, 5, 2]
>>> g = (n * n for n in nums)
>>> g
```

```
<generator object <genexpr> at 0x7fd90e7b22e0>
>>> next(g)
10000

# here's a generator version of the sum_sqr_evens(iterable) function
# note that () is optional here for the generator expression
>>> sum(n * n for n in nums if n % 2 == 0)
11028

# inner product
>>> sum(i * j for i, j in zip((1, 3, 5), (2, 4, 6)))
44
```

Here's an example with the `join()` method:

```
>>> items = (1, 'hi', [10, 20], 'bye')
>>> ':'.join(items)
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    ':'.join(items)
    ~~~~~~~~^^^^^^^
TypeError: sequence item 0: expected str instance, int found
>>> ':'.join(str(i) for i in items)
'1:hi:[10, 20]:bye'
```

Exercises

- Write a function that returns a dictionary sorted by values in ascending order.

  ```
  >>> marks = dict(Rahul=86, Ravi=92, Rohit=75, Rajan=79, Ram=92)
  >>> sort_by_value(marks)
  {'Rohit': 75, 'Rajan': 79, 'Rahul': 86, 'Ravi': 92, 'Ram': 92}
  ```

- Write a function that returns a `list` of string slices as per the following rules:

  - return the input string as the only element if its length is less than 3 characters
  - otherwise, return all slices that have 2 or more characters

  ```
  >>> word_slices('')
  ['']
  >>> word_slices('i')
  ['i']
  >>> word_slices('to')
  ['to']
  >>> word_slices('table')
  ['ta', 'tab', 'tabl', 'table', 'ab', 'abl', 'able', 'bl', 'ble', 'le']
  ```

- Square even numbers and cube odd numbers. For example, `[321, 1, -4, 0, 5, 2]` should give `[33076161, 1, 16, 0, 125, 4]` as the output.

- Calculate sum of squares of the numbers, only if the square value is less than `50` . Output for `(7.1, 1, -4, 8, 5.1, 12)` should be `43.01` .

Dealing with files

This chapter will discuss the `open()` built-in function and introduce some of the built-in modules for file processing.

open and close

The open() built-in function is one of the ways to read and write files. The first argument to this function is the filename (relative or absolute path) to be processed. Rest are keyword arguments that you can configure. The output is a `TextIOWrapper` object (filehandle), which you can use as an iterator. Here's an example:

```
# default mode is rt (read text)
>>> fh = open('ip.txt')
>>> fh
<_io.TextIOWrapper name='ip.txt' mode='r' encoding='UTF-8'>
>>> next(fh)
'hi there\n'
>>> next(fh)
'today is sunny\n'
>>> next(fh)
'have a nice day\n'
>>> next(fh)
Traceback (most recent call last):
  File "<python-input-5>", line 1, in <module>
    next(fh)
    ~~~~^^^^
StopIteration

# check if the filehandle is active or closed
>>> fh.closed
False
# close the filehandle
>>> fh.close()
>>> fh.closed
True
```

The `mode` argument specifies what kind of processing you want. Only the `text` mode will be covered in this chapter, which is the default. You can combine options — for example, `rb` means `read` in `binary` mode. Here are the relevant details from the documentation:

- `r` open for reading (default)
- `w` open for writing, truncating the file first
- `x` open for exclusive creation, failing if the file already exists
- `a` open for writing, appending to the end of the file if it exists
- `b` binary mode
- `t` text mode (default)
- `+` open for updating (reading and writing)

The `encoding` argument is meaningful only in the `text` mode. You can check the default encoding for your environment using the locale module as shown below. See docs.python: standard encodings and docs.python HOWTOs: Unicode for more details.

```
>>> import locale
>>> locale.getpreferredencoding()
'UTF-8'
```

Here's how Python handles line separation by default, see documentation for more details.

> When reading input from the stream, if `newline` is `None`, universal newlines mode is enabled. Lines in the input can end in `'\n'`, `'\r'`, or `'\r\n'`, and these are translated into `'\n'` before being returned to the caller.

> When writing output to the stream, if `newline` is `None`, any `'\n'` characters written are translated to the system default line separator, `os.linesep`.

If the given filename doesn't exist, you'll get a `FileNotFoundError` exception.

```
>>> open('xyz.txt', mode='r', encoding='ascii')
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    open('xyz.txt', mode='r', encoding='ascii')
    ~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'xyz.txt'
```

Context manager

Quoting from docs.python: Reading and Writing Files:

> It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using `with` is also much shorter than writing equivalent `try-finally` blocks.

Here's an example:

```
# read_file.py
with open('ip.txt', mode='r', encoding='ascii') as f:
    for ip_line in f:
        op_line = ip_line.rstrip('\n').capitalize() + '.'
        print(op_line)
```

Recall that `as` keyword was seen before in the Different ways of importing and try-except sections. Here's the output of the above program:

```
$ python3.13 read_file.py
Hi there.
Today is sunny.
Have a nice day.
```

> See The Magic of Python Context Managers for more examples and details.

read, readline and readlines

The `read()` method gives you the entire remaining contents of the file as a single string. The `readline()` method gives the next line of text and `readlines()` gives all the remaining lines as a `list` of strings.

```
>>> open('ip.txt').read()
'hi there\ntoday is sunny\nhave a nice day\n'

>>> fh = open('ip.txt')
# readline() is similar to next()
# but returns empty string instead of StopIteration exception
>>> fh.readline()
'hi there\n'
>>> fh.readlines()
['today is sunny\n', 'have a nice day\n']
>>> fh.readline()
''
```

write

```
# write_file.py
with open('op.txt', mode='w', encoding='ascii') as f:
    f.write('this is a sample line of text\n')
    f.write('yet another line\n')
```

You can call the `write()` method on a filehandle to add contents to that file (provided the `mode` you have set supports writing). Unlike `print()`, the `write()` method doesn't automatically add newline characters.

```
$ python3.13 write_file.py

$ cat op.txt
this is a sample line of text
yet another line

$ file op.txt
op.txt: ASCII text
```

⚠ If the file already exists, the `w` mode will overwrite the contents (i.e. any existing content will be lost).

You can also use the `print()` function for writing by passing the filehandle to the `file` keyword argument. The fileinput module supports in-place editing and other features (see the In-place editing with fileinput section for examples).

File processing modules

This section gives introductory examples for some of the built-in modules that are handy for file processing. Quoting from docs.python: os:

This module provides a portable way of using operating system dependent functionality.

```
>>> import os

# current working directory
>>> os.getcwd()
'/home/learnbyexample/Python/programs/'

# value of an environment variable
>>> os.getenv('SHELL')
'/bin/bash'

# file size
>>> os.stat('ip.txt').st_size
40

# check if the given path is a file
>>> os.path.isfile('ip.txt')
True
```

Quoting from docs.python: glob:

The glob module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but `*` , `?` , and character ranges expressed with `[]` will be correctly matched.

```
>>> import glob

# list of files (including directories) containing '_file' in their name
>>> glob.glob('*_file*')
['read_file.py', 'write_file.py']
```

Quoting from docs.python: shutil:

The `shutil` module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal.

```
>>> import shutil

>>> shutil.copy('ip.txt', 'ip_file.txt')
'ip_file.txt'
>>> glob.glob('*_file*')
['read_file.py', 'ip_file.txt', 'write_file.py']
```

Quoting from docs.python: pathlib:

This module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between pure paths, which provide purely computational operations without I/O, and concrete paths, which inherit from pure paths but also provide I/O operations.

```
>>> from pathlib import Path

# use 'rglob' instead of 'glob' if you want to match names recursively
>>> list(Path('programs').glob('*file.py'))
[PosixPath('programs/read_file.py'), PosixPath('programs/write_file.py')]
```

See Python pathlib Cookbook, pathlib module tips and tricks and stackoverflow: How can I iterate over files in a given directory? for more details and examples.

There are specialized modules for structured data processing as well, for example:

- docs.python: csv
- docs.python: json
- docs.python: xml

Exercises

- Write a program that reads a known filename `f1.txt` which contains a single column of numbers. Your task is to display the sum of these numbers, which is `10485.14` for the given example.

  ```
  $ cat f1.txt
  8
  53
  3.14
  84
  73e2
  100
  2937
  ```

- Read the documentation for `glob.glob()` and write a program to list all files ending with `.txt` in the current directory as well as sub-directories, recursively.

Executing external commands

This chapter will show how to execute external commands from Python, capture their output and other relevant details such as the exit status. The availability of commands depends on the OS you are using (mine is Linux).

os module

Last chapter showed a few examples with the `os` module for file processing. This module is useful in plenty of other situations as well — for example, providing an interface for working with external commands.

```
>>> import os

>>> os.system('echo hello "$USER"')
hello learnbyexample
0
```

Similar to the `print()` function, the output of the external command, if any, is displayed on the screen. The return value is the exit status of the command, which gets displayed by default on the REPL. `0` implies that the command executed successfully, any other value indicates some kind of failure. As per docs.python: os.system:

> On Unix, the return value is the exit status of the process encoded in the format specified for `wait()` .

Here's an example with non-zero exit status:

```
>>> status = os.system('ls xyz.txt')
ls: cannot access 'xyz.txt': No such file or directory
>>> status
512
# to get the actual exit value
>>> os.waitstatus_to_exitcode(status)
2

# redirect the stderr stream if you don't want to see the error message
>>> os.system('ls xyz.txt 2> /dev/null')
512
```

You can use the `os.popen()` method to save the results of an external command. It provides a file object like interface for both read (default) and write. To check the status, call the `close()` method on the filehandle (`None` means success).

```
>>> fh = os.popen('wc -w <ip.txt')
>>> op = fh.read()
>>> op
'9\n'
>>> status = fh.close()
>>> print(status)
None
```

```
# if you just want the output
>>> os.popen('wc -w <ip.txt').read()
'9\n'
```

subprocess.run

The `subprocess` module provides a more flexible and secure option to execute external commands, at the cost of being more verbose.

Quoting relevant parts from doc.python: subprocess module:

> The `subprocess` module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

> The recommended approach to invoking `subprocesses` is to use the `run()` function for all use cases it can handle. For more advanced use cases, the underlying `Popen` interface can be used directly.

```
>>> import subprocess

>>> subprocess.run('pwd')
'/home/learnbyexample/Python/programs/'
CompletedProcess(args='pwd', returncode=0)

>>> process = subprocess.run(('ls', 'xyz.txt'))
ls: cannot access 'xyz.txt': No such file or directory
>>> process.returncode
2
```

The first argument to the `run()` method is the command to be executed. This can be either a single string or a sequence of strings (if you need to pass arguments to the command being executed). By default, the command output is displayed on the screen. A `CompletedProcess` object is returned, which has relevant information for the command that was executed such as the exit status.

As an **exercise**, read the subprocess.run documentation and modify the above `ls` example to:

- redirect the `stderr` stream to `/dev/null`
- automatically raise an exception when the exit status is non-zero

See also:

- stackoverflow: How to execute a program or call a system command from Python?
- stackoverflow: difference between subprocess and os.system
- stackoverflow: How to use subprocess command with pipes
- stackoverflow: subprocess FAQ

shell=True

You can also construct a single string command, similar to `os.system()` , if you set the `shell` keyword argument to `True` . While this is convenient, use it only if you have total control over the command being executed such as your personal scripts. Otherwise, it can lead to security issues, see stackoverflow: why not use shell=True for details.

Quoting from docs.python: subprocess Frequently Used Arguments:

> If `shell` is `True` , the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of `~` to a user's home directory.

Here are some examples:

```
>>> p = subprocess.run(('echo', '$HOME'))
$HOME
>>> p = subprocess.run('echo $HOME', shell=True)
/home/learnbyexample

>>> p = subprocess.run(('ls', '*.txt'))
ls: cannot access '*.txt': No such file or directory
>>> p = subprocess.run('ls *.txt', shell=True)
ip.txt

>>> p = subprocess.run('seq -s, 10 > out.txt', shell=True)
>>> p = subprocess.run('cat out.txt', shell=True)
1,2,3,4,5,6,7,8,9,10
```

If `shell=True` cannot be used but shell features as quoted above are needed, you can use modules like `os` , `glob` , `shutil` and so on as applicable. See also docs.python: Replacing Older Functions with the subprocess Module.

```
>>> p = subprocess.run(('echo', os.getenv('HOME')))
/home/learnbyexample
```

Changing shell

By default, `/bin/sh` is the shell used for POSIX systems. You can change that by setting the `executable` argument to the shell of your choice.

```
>>> p = subprocess.run('diff <(seq 3) <(seq 4)', shell=True)
/bin/sh: 1: Syntax error: "(" unexpected

>>> p = subprocess.run('diff <(seq 3) <(seq 4)', shell=True,
                       executable='/bin/bash')
3a4
> 4
```

Capture output

If you use `capture_output=True`, the `CompletedProcess` object will provide `stdout` and `stderr` results as well. These are provided as the `bytes` data type by default. You can change that by setting `text=True`.

```
>>> p = subprocess.run(('date', '-u', '+%A'), capture_output=True, text=True)
>>> p
CompletedProcess(args=('date', '-u', '+%A'), returncode=0,
                 stdout='Monday\n', stderr='')
>>> p.stdout
'Monday\n'
```

You can also use `subprocess.check_output()` method to directly get the output.

```
>>> subprocess.check_output(('date', '-u', '+%A'), text=True)
'Monday\n'
```

You can also use the legacy methods `subprocess.getstatusoutput()` and `subprocess.getoutput()` but they lack in features and do not provide secure options. See docs.python: subprocess Legacy Shell Invocation Functions for details.

Command line arguments

This chapter will show a few examples of processing CLI arguments using the `sys` and `argparse` modules. The `fileinput` module is also introduced in this chapter, which is handy for in-place file editing.

sys.argv

Command line arguments passed when executing a Python program can be accessed as a `list` of strings via `sys.argv`. The first element (index `0`) contains the name of the Python script or `-c` or empty string, depending upon how the Python interpreter was called. Rest of the elements will have the command line arguments, if any were passed along the script to be executed. See docs.python: sys.argv for more details.

Here's a program that accepts two numbers passed as CLI arguments and displays the sum only if the input was passed correctly.

```
# sum_two_nums.py
import ast
import sys

try:
    num1, num2 = sys.argv[1:]
    total = ast.literal_eval(num1) + ast.literal_eval(num2)
except ValueError:
    sys.exit('Error: Please provide exactly two numbers as arguments')
else:
    print(f'{num1} + {num2} = {total}')
```

The ast.literal_eval() method is handy for converting a string value to built-in literals, especially for collection data types. If you wanted to use `int()` and `float()` for the above program, you'd have to add logic for separating the input into integers and floating-point first. Passing a string to sys.exit() gets printed to the `stderr` stream and sets the exit status as `1` in addition to terminating the script.

Here's a sample run:

```
$ python3.13 sum_two_nums.py 2 3.14
2 + 3.14 = 5.140000000000001
$ echo $?
0

$ python3.13 sum_two_nums.py 2 3.14 7
Error: Please provide exactly two numbers as arguments
$ echo $?
1
$ python3.13 sum_two_nums.py 2 abc
Error: Please provide exactly two numbers as arguments
```

As an **exercise**, modify the above program to handle `TypeError` exceptions. Instead of the output shown below, inform the user about the error using the `sys.exit()` method.

```
$ python3.13 sum_two_nums.py 2 [1]
Traceback (most recent call last):
  File "/home/learnbyexample/Python/programs/sum_two_nums.py", line 6, in <module>
    total = ast.literal_eval(num1) + ast.literal_eval(num2)
            ~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

As another **exercise**, write a program that accepts one or more numbers as input arguments. Calculate and display the following details about the input — sum, product and average.

In-place editing with fileinput

To edit a file in-place, the fileinput module comes in handy. Here's a program that loops over filenames passed as CLI arguments (i.e. `sys.argv[1:]`), does some processing and writes back the changes to the original input files. You can also provide one or more filenames to the `files` keyword argument (helpful when the file list is pre-determined or generated programmatically).

```python
# inplace_edit.py
import fileinput

with fileinput.input(inplace=True) as f:
    for ip_line in f:
        op_line = ip_line.rstrip('\n').capitalize() + '.'
        print(op_line)
```

Note that unlike `open()` , the `FileInput` object doesn't support the `write()` method. Using the `print()` function is enough as shown above. Here's a sample run:

```
$ python3.13 inplace_edit.py [io]p.txt

# check if the files have changed
$ cat ip.txt
Hi there.
Today is sunny.
Have a nice day.
$ cat op.txt
This is a sample line of text.
Yet another line.

# if stdin is passed as input, inplace gets disabled
$ echo 'GooD moRNiNg' | python3.13 inplace_edit.py
Good morning.
```

As `inplace=True` permanently modifies your input files, it is always a good idea to check your logic on sample files first. That way your data wouldn't be lost because of an error in your program. You can also ask `fileinput` to create backups if you need to recover original files later — for example, the keyword argument `backup='.bkp'` will create backups by adding `.bkp` as the suffix to the original filenames.

argparse

`sys.argv` is good enough for simple use cases. If you wish to create a CLI application with various kinds of flags and arguments (some of which may be optional or mandatory) and so on, use a module such as the built-in `argparse` or a third-party solution like click.

Quoting from docs.python: argparse:

> The `argparse` module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and `argparse` will figure out how to parse those out of `sys.argv`. The `argparse` module also automatically generates help and usage messages. The module will also issue errors when users give the program invalid arguments.

Here's a CLI application that accepts a file containing a list of filenames that are to be sorted by their extension. Files with the same extension are further sorted in ascending order. The program also implements an optional flag to remove duplicate entries.

```python
# sort_ext.py
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-f', '--file', required=True,
                    help="input file to be sorted")
parser.add_argument('-u', '--unique', action='store_true',
                    help="sort uniquely")
args = parser.parse_args()

ip_lines = open(args.file).readlines()
if args.unique:
    ip_lines = set(ip_lines)

op_lines = sorted(ip_lines, key=lambda s: (s.rsplit('.', 1)[-1], s))
for line in op_lines:
    print(line, end='')
```

The documentation for the CLI application is generated automatically based on the information passed to the parser. You can use the help option (which is added automatically too) to view the documentation, as shown below:

```
$ python3.13 sort_ext.py -h
usage: sort_ext.py [-h] -f FILE [-u]

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  input file to be sorted
  -u, --unique          sort uniquely

$ python3.13 sort_ext.py
usage: sort_ext.py [-h] -f FILE [-u]
sort_ext.py: error: the following arguments are required: -f/--file
```

The `add_argument()` method allows you to add details about an option/argument for the CLI application. The first parameter names an argument or option (starts with `-`). The `help` keyword argument lets you add documentation for that particular option/argument. See docs.python: add_argument for documentation and details about other keyword arguments.

The above program adds two options, one to store the filename to be sorted and the other to act as a flag for sorting uniquely. Here's a sample text file that needs to be sorted based on the extension.

```
$ cat sample.txt
input.log
basic.test
input.log
out.put.txt
sync.py
input.log
async.txt
```

Here's the output with both types of sorting supported by the program.

```
# default sort
$ python3.13 sort_ext.py -f sample.txt
input.log
input.log
input.log
sync.py
basic.test
async.txt
out.put.txt

# unique sort
$ python3.13 sort_ext.py -uf sample.txt
input.log
sync.py
basic.test
async.txt
out.put.txt
```

See docs.python HOWTOs: Argparse Tutorial for a more detailed lesson.

Accepting stdin

CLI tools like `grep`, `sed`, `awk` and many others can accept data from `stdin` as well as accept filenames as arguments. The previous program modified to add `stdin` functionality is shown below.

`args.file` is now a positional argument instead of an option. `nargs='?'` indicates that this argument is optional. `type=argparse.FileType('r')` allows you to automatically get a filehandle in `read` mode for the filename supplied as an argument. If filename isn't provided, `default=sys.stdin` kicks in and you get a filehandle for the `stdin` data.

```
# sort_ext_stdin.py
import argparse, sys

parser = argparse.ArgumentParser()
parser.add_argument('file', nargs='?',
                    type=argparse.FileType('r'), default=sys.stdin,
                    help="input file to be sorted")
parser.add_argument('-u', '--unique', action='store_true',
                    help="sort uniquely")
args = parser.parse_args()

ip_lines = args.file.readlines()
if args.unique:
    ip_lines = set(ip_lines)

op_lines = sorted(ip_lines, key=lambda s: (s.rsplit('.', 1)[-1], s))
for line in op_lines:
    print(line, end='')
```

Here's the help for the modified program:

```
$ python3.13 sort_ext_stdin.py -h
usage: sort_ext_stdin.py [-h] [-u] [file]

positional arguments:
  file          input file to be sorted

optional arguments:
  -h, --help    show this help message and exit
  -u, --unique  sort uniquely
```

Here's a sample run showing both `stdin` and filename argument functionality.

```
# 'cat' is used here for illustration purposes only
$ cat sample.txt | python3.13 sort_ext_stdin.py
input.log
input.log
input.log
sync.py
basic.test
async.txt
out.put.txt
$ python3.13 sort_ext_stdin.py -u sample.txt
input.log
sync.py
basic.test
async.txt
out.put.txt
```

As an **exercise**, add an optional argument `-o, --output` to store the output in a file for the above program.