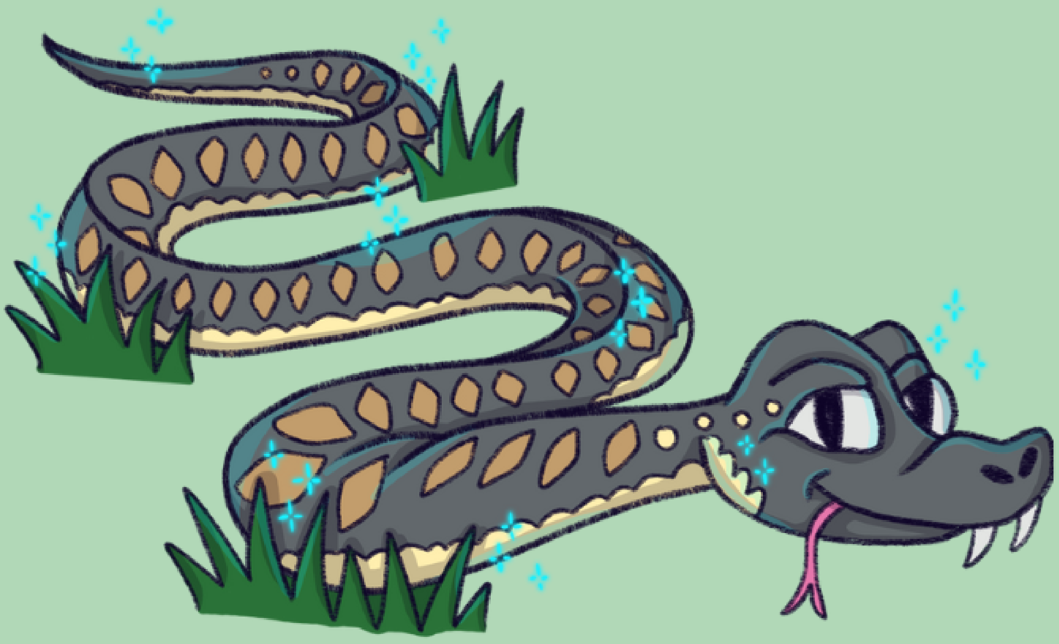
learnbyexample

100 Page Python Intro



Sundeep Agarwal

Table of contents

Preface 4

Prerequisites . . . 4
Conventions . . . 4
Acknowledgements . . . 4
Feedback and Errata . . . 5
Author info . . . 5
License . . . 5
Book version . . . 5

Introduction 6

Installation . . . 6
Online tools . . . 6
First program . . . 7
IDE and text editors . . . 7
REPL . . . 8
Documentation and getting help . . . 9

Numeric data types 11

int . . . 11
float . . . 11
Arithmetic operators . . . 12
Operator precedence . . . 13
Integer formats . . . 13
Other numeric types . . . 14

Strings and user input 15

Single and double quoted strings . . . 15
Triple quoted strings . . . 16
Raw strings . . . 16
String operators . . . 17
String formatting . . . 18
User input . . . 20
Type conversion . . . 20
Exercises . . . 21

Defining functions 22

def . . . 22
Accepting arguments . . . 22
Default valued arguments . . . 23
Return value . . . 24
A closer look at the print() function . . . 25
Docstrings . . . 27
Interactive TUI app for exercises . . . 27

Control structures 29

Comparison operators . . . 29
Truthy and Falsy values . . . 29
Boolean operators . . . 30
Comparison chaining . . . 31

Membership operator 31
if-elif-else 32
Ternary operator 32
for loop 33
while loop 34
break and continue 34
Assignment expression 35
Exercises 36

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.

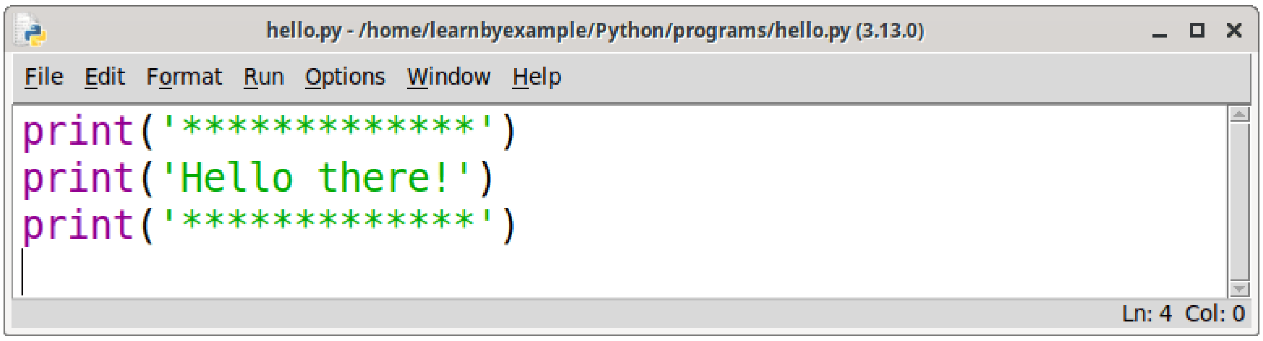
See Python behind the scenes and this list of resources if you are interested to learn inner details about Python program execution.

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

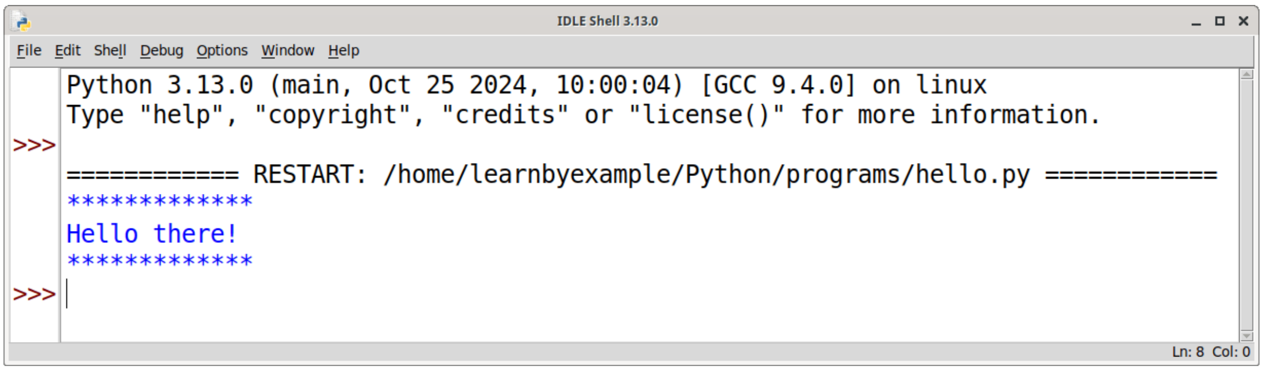
An **integrated development environment** (IDE) might suit you better if you are not comfortable with the command line. IDE provides features likes debugging, syntax highlighting, autocompletion, code refactoring and so on. They also help in setting up a **virtual environment** to manage different versions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

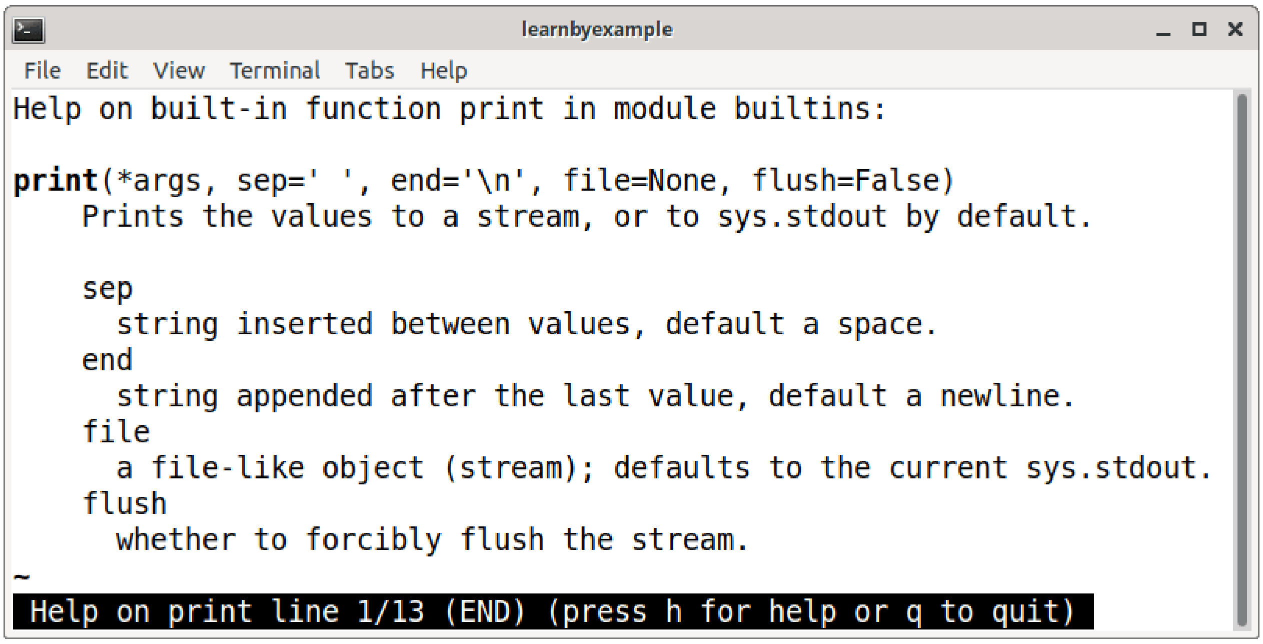
Unlike previous versions, the Python REPL now implements editing and navigation features on its own instead of relying on an external `readline` library. See REPL-acing the default REPL (PEP 762) for more information.

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 stack-overflow: 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.

```python
# 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`).

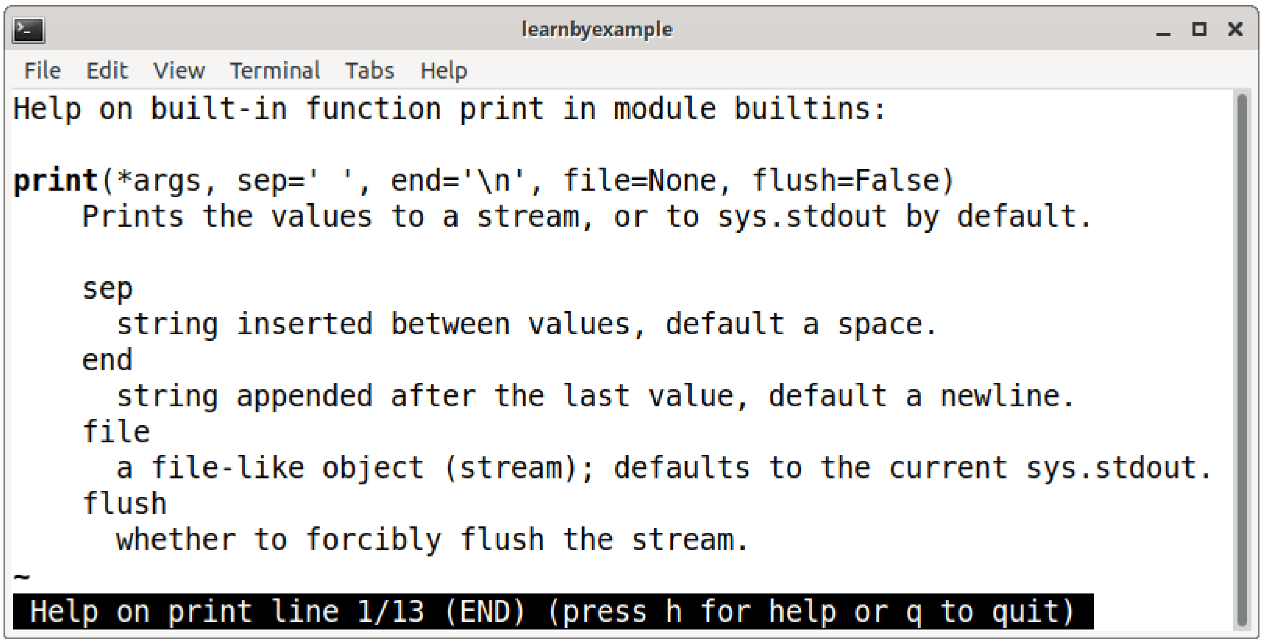
```
>>> def num_square(n):
...     return n * n
...
>>> num_square(5)
25
>>> num_square(3.14)
9.8596

>>> 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)
```

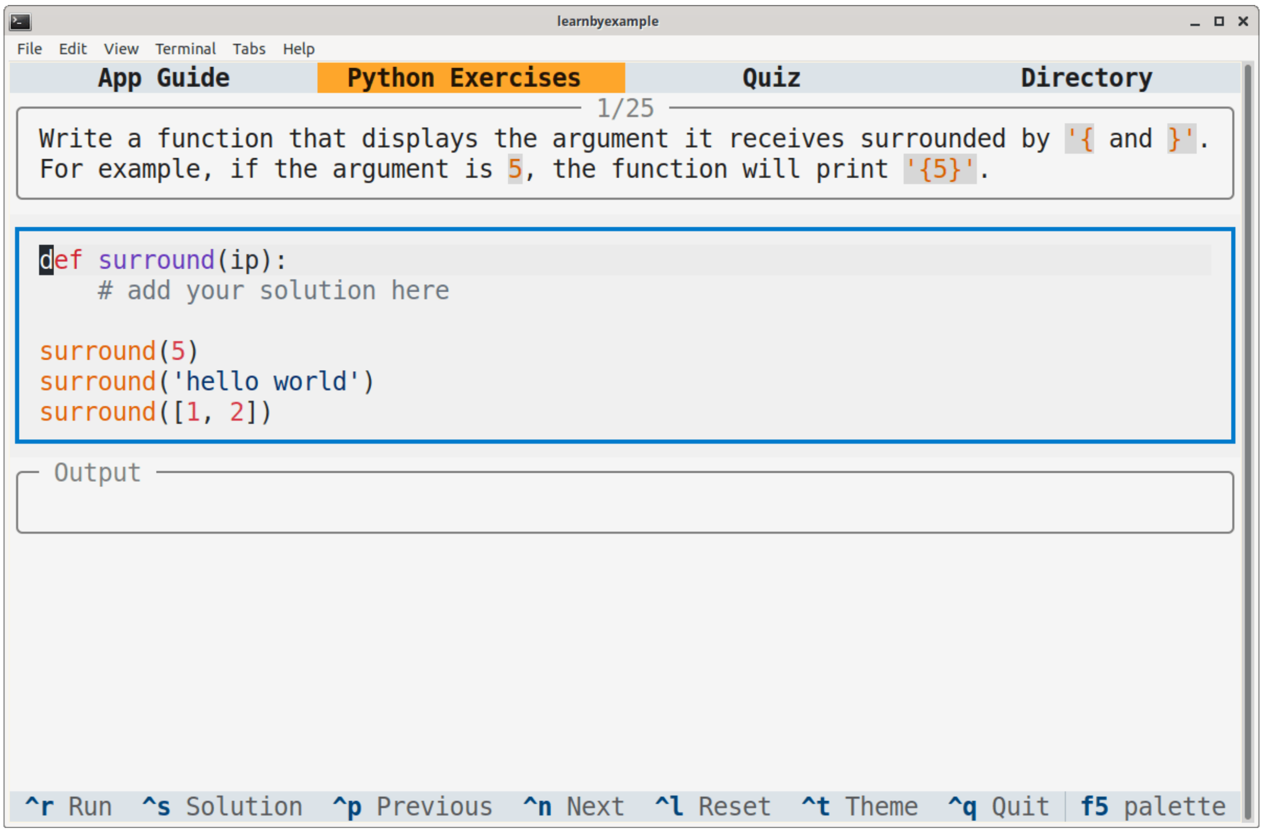
Calling `help(num_square)` will give you the documentation as shown below.

```
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}'.

```python
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:

```python
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:

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

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

Here's a sample multiplication table:

```python
>>> 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.

```python
>>> 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.