The Author
Mike Chyson

The Big Book of

PYTHON 3

Notebook on programming in python 3

Jan, 15, 2021

Dedication

Notebook on “Programming in Python 3” second edition.
Learn a programming language with:

- Think and summerize.
- Programme.

The code is in `https://github.com/mikechyson/python3`

Contents

Contents 1

1 Introduction 5

1.1 Creating and running Pyton programs 5
1.2 Python's "Beautiful Heart" 6
1.2.1 Data types 6
1.2.2 Object references 6
1.2.3 Collection data types 7
1.3 Logical operations 8
1.3.1 The identity operator 9
1.3.2 Comparison operators 9
1.3.3 The membership operator 10
1.3.4 Logical operators 10
1.4 Control flow statements 11
1.4.1 The if statement 12
1.4.2 The while statement 12
1.4.3 The for ... in statement 12
1.4.4 Basic exception handling 12
1.5 Arithmetic operators 13
1.6 Input/output 14
1.7 Creating and calling functions 15

2 Data types 17

2.1 Identifiers and keywords 17
2.2 Integral types 18
2.2.1 Integers 18
2.2.2 Booleans 20

2.3 Floating-point types 20
2.3.1 Floating-point numbers 21
2.3.2 Complex numbers 22
2.3.3 Decimal numbers 22
2.4 Strings 22
2.4.1 Comparing strings 24
2.4.2 Slicing and striding strings 24
2.4.3 String operators and methods 25
2.4.4 String formatting with the str.format() method 27
2.4.5 Field names 27
2.4.6 conversions 28
2.4.7 Format specifications 29

3 Collection data types 33
3.1 Sequence types 33
3.1.1 Tuples 34
3.1.2 Named tuples 34
3.1.3 Lists 35
3.1.4 List comprehensions 36
3.2 Set types 37
3.2.1 Sets 37
3.2.2 Set comprehensions 39
3.2.3 Frozen sets 39
3.3 Mapping types 39
3.3.1 Dictionaries 40
3.3.2 Dictionary comprehensions 41
3.3.3 Default dictionaries 42
3.3.4 Ordered dictionaries 42
3.4 Iterating and copying collections 43
3.4.1 Iterators and iterable operations and functions 43
3.4.2 Copying collections 45

4 Control structures and functions 47
4.1 Control structures 47
4.1.1 Conditional branching 47
4.1.2 Looping 48
4.2 Exception handling 49

4.2.1 Catching and raising exceptions 49
4.2.2 Custom exceptions 50
4.3 Costom functions 53
4.3.1 Names and docstrings 55
4.3.2 Argument and parameter unpacking 56
4.3.3 Accessing variables in the global scope 57
4.3.4 Lambda functions 58
4.3.5 Assertions 58

Chapter 1

Introduction

1.1 Creating and running Pyton programs

Be default, Python files are assumed to use the UTF-8 character encoding. Python files normally have an extension of `.py`. Python GUI (Graphical User Interface) programs usually have an extension of `.pyw`.

```
#!/usr/bin/env python3

print("Hello", "world")
```

The first line is a comment. In Python,comments begin with a `#` and continue to the end of the line. The second line is blank. Python ignores blank lines, but they are often useful to humans to break up large blocks of code to make them easier to read. The third line is Python code.

Each statement encountered in a `.py` file is executed in turn, starting with the first one and progressing line by line. Python programs are executed by the Python interpreter, and normally this is done inside a console window.

On Unix, when a program is invoked in the console, the file's first two bytes are read. If these bytes are the ASCII chracters `#!`, the shell assume that the file is to be executed by an interpreter and that the file's first line specifies which interpreter to use. This line is called the **shebang** (shell execute) line, and if the present must be the first line in the file.

The shebang line is commonly written in one of two forms, either:

```
#!/usr/bin/python3
```

or

```
#!/usr/bin/env python3
```

If written using the first form, the specified interpreter is used. If written using the second form, the first `python3` interpreter found in the shell's current environment is used. The second form is more versatile because it allows for the possibility that the Python 3 interpreter is ont located in `/usr/bin`

1.2 Python's "Beautiful Heart"

1.2.1 Data types

One fundamental thing that any programming language must be able to do is represent items of data.

The size of Python's integers is limited only by machine memory, not by a fixed number of bytes. Strings can be delimited by double or single quotes, as long as the same kind are used at both ends.

```
2106245833371143733958360553673408646377901908010982225086219550 72
0
"hello"
'world'
```

Python uses square brackets ([]) to access an item from a sequence such as a string.

```
'Hello World'[4]
```

In Python, both `str` and the basic numeric types such as `int` are **immutable**. At first this appears to be a rather strange limitation, but Python's syntax means that this is a nonissue in practice. The only reason for mentioning it is that although we can use square brackets to retrieve the character at a given index position in a string, we cannot use them to set a new character.

To convert a data item from one type to another we can use the syntax `datatype(item)`.

```
int("45")
str(123)
```

1.2.2 Object references

Once we have some data types, the next thing we need are variables in which to store them. Python doesn't have variables as such, but instead has **object references**. When it comes to immutable objects like `int`s and `str`s, there is no discernable difference between a variable and an object reference. As for mutable objects, there is a difference, but it rarely matters in practice.

```

x = "blue"
y = "green"
z = x
```

The syntax is simply `object_reference = value`. The `=` operator is not the same as the variable assignment operater in some other languages. The `=` operater binds an object reference to an object in memory. If the object reference already exists, it is simply re-bound to refer to the object on the right of the `=` operator; if the object reference does not exist it is created by the `=` operator.

Python uses **dynamic typing**, which means that an object reference can be rebound to refer to a different object at any time. Languages that use strong typing (such as C++ and Java) allow only those operations that are defined for the data types involved to be performed. Python also applies this constraint, but it isnt called strong typing in Pythons case because the valid operations can change — for example, if an object reference is re-bound to an object of a different data type.

```

route = 123
print(route, type(route))

route = "North"
print(route, type(route))
```

The `type()` function returns the data type (also known as the "class") of the data item it is given — this function can be very useful for testing and debugging, but would not normally appear in production code.

1.2.3 Collection data types

To hold entire collections of data items, Python provides several collection data types that can hold items. Python tuples and lists can be used to hold any number of data items of any data types. Tuples are imuutable while lists are mutable.

Tuples are created using commas (,), as these examples show:

```

>>> "hello", "world", "mike", "chyson"
("hello", "world", "mike", "chyson")
>>> "one",
("one",)
```

When Python ouptuts a tuple it encloses it in parentheses. An empty tuple is created by using empty parentheses, (). The comma is also used to separate arguments in function calls, so if we want to pass a tuple literal as an argument we must enclose it in parentheses to avoid confusion.

```

[1,2,3]
[]
```

One way to create a list is to use square brackets ([]). An empty list is create by using empty brackets, [].

Under the hood, lists and tuples don't store data items at all, but rather object references. When lists and tuples are created (and when items are inserted in the case of lists), they take copies of the object references they are given. In the case of literal items such as integers or strings, an object of the appropriate data type is created in memory and suitably initialized, and then an object reference referring to the object is created, and it is this object reference that is put in the list or tuple.

In precedural programming we can function and often pass in data items as arguments.

```

>>> len(("one",))
1
>>> len([1, 2, "hell", 3])
4
```

All Python data items are **objects** (also called **instances**) of a particular data type (also called a class).

```

>>> x = ["zebra", 49, -879, "aardvark", 200]
>>> x.append("more")
>>> x
['zebra', 49, -879, 'aardvark', 200, 'more']

>>> list.append(x, "extra")
>>> x
['zebra', 49, -879, 'aardvark', 200, 'more', 'extra']
```

Python has conventional functions called like this `function_name(arguments)`; and methods which are called like this `ojbect_name.method_name(arguments)`.

The dot ("access attribute") operator is used to access an object's attributes.

1.3 Logical operations

Python provides four sets of logical operations.

1.3.1 The identity operator

The `is` operator is a binary operator that returns `True` if its left-hand object reference is referencing to the same object as its right-hand object reference.

```
>>> a = ["Retention", 3, None]
>>> b = ["Retention", 3, None]
>>> a is b
False
>>> b = a
>>> a is b
True
```

One benefit of identity comparisons is that they are very fast. This is because the objects referred to do not have to be examined themselves. The is operator needs to compare only the memory addresses of the objects — the same address means the same object.

The most common use case for `is` is to compare a data item with the built-in null object, `None`.

The purpose of the identity operator is to see whether two object references refer to the same object, or to see whether an object is `None`. If we want to compare object values we should use a comparison operator instead.

1.3.2 Comparison operators

Python provides the standard set of binary comparison operators:

- <
- <=
- ==
- !=
- >=
- >

These operators compare object values, that is, objects that the object references used in the comparison refer to.

In some cases, comparing the identity of two strings or numbers will return `True`, even if each has been assigned separately. This is because some implementations of Python will reuse the same object (since the value is the same and is immutable) for the sake of efficiency. The moral of this is to use `==` and

`!=` when comparing **`values`**, and to use `is` and `is not` only when comparing with `None` or when we really do want to see if two object references, rather than their values, are the same.

One particularly nice feature of Pythons comparison operators is that they can be chained. For example:

```

>>> a = 9
>>> 0 <= a <= 10
True
```

This is a nicer way of testing that a given data item is in range than having to do two separate comparisons joined by logical `and`. It also has the additional virtue of evaluating the data item only once (since it appears once only in the expression), something that could make a difference if computing the data item's value is expensive, or if accessing the data item causes side effects.

1.3.3 The membership operator

For data types that are sequences or collections such as strings, lists, and tuples, we can test for membership using the `in` operator, and for nonmembership using the `not in` operator. For example:

```

>>> p = (4, "frog", 9, -33, 9, 2)
>>> 2 in p
True
>>> "dog" not in p
True
```

For lists and tuples, the `in` operator uses a linear search which can be slow for very large collections (tens of thousands of items or more). On the other hand, `in` is very fast when used on a dictionary or a set.

1.3.4 Logical operators

Python provides 3 logical operators:

- and
- or
- not

Both `and` and `or` use short-circuit logic and return the operand that determined the result – they do not return a Boolean (unleass they actually have Boolean operands).

```

>>> five = 5
>>> two = 2
>>> zero = 0
>>> five and two
2
>>> two and five
5
>>> five and zero
0
```

If the expression occurs in a Boolean context, the result is evaluated as a Boolean, so the preceding expressions would come out as `True`, `True`, and `False`.

```

>>> nought = 0
>>> five or two
5
>>> two or five
2
>>> zero or five
5
>>> zero or nought
0
```

The `or` operator is similar; here the results in a Boolean context would be `True`, `True`, `True`, and `False`.

The `not` unary operator evaluates its argument in a Boolean context and always returns a Boolean result.

1.4 Control flow statements

A Boolean expression is anything that can be evaluated to produce a Boolean value (`True` or `False`). In Python, such an expression evaluate to `False` if it is the predefined constant `False`, the special object `None`, an empty sequence or collection, or a numeric data item of value 0; anything else is considered to be `True`.

In Python-speak a block of code, that is, a sequence of one or more statements, is called a **suite**. Because some of Python's syntax requires that a suite be present, Python provides the keyword `pass` which is a statement that does nothing and that can be used where a suite is required but where no precessing is necessary.

1.4.1 The if statement

```
if boolean_expression1:
    suite1
elif boolean_expression2:
    suite2
...
elif boolean_expressionN:
    suiteN
else:
    else_suite
```

Colons are used with `else`, `elif`, and essentially in any other place where a suite is to follow. Unlike most other programming languages, Python uses indentation to signify its block structure.

1.4.2 The while statement

```
while boolean_expression:
    suite
```

1.4.3 The for ... in statement

```
for variable in iterable:
    suite
```

The `variable` is set to refer to each object in the `iterable` in turn.

1.4.4 Basic exception handling

An exception is an object like any other Python object, and when converted to a string, the exception produces a message text. A simple form of the syntax for exception handlers is this:

```
try:
    try_suite
except exceptions1 as variable1:
    exception_suite1
...
except exceptionN as variableN:
    exception_suiteN
```

The `as variable` part is optional.

```
s = input("enter an integer: ")
try:
    i = int(s)
    print("valid integer entered:", i)
except ValueError as err:
    print(err)
```

1.5 Arithmetic operators

Four basic mathematical operations:

- +
- -
- *
- /

In addition, many Python data types can be used with augmented assignment operators such as:

- +=
- -=
- *=
- /=

The +, -, and * operators all behave as expected when both of their operands are integers. Where Python differs from the crowd is when it comes to division:

```

>>> 12/3
4.0
```

The division operator produces a floating-point value, not a integer. If we nned an integer result, we can always convert using `int()` or use the truncating devision operator `//`.

```

>>> a = 5
>>> a
5
>>> a += 8
>>> a
13
```

Comparing to C-like languages, there are two important subtleties, one Python-specific and one to do with augmented operators in any language.

The first point to remember is that the `int` data type is immutable. So, what actually happens behind the scenes when an augmented assignment operator is used on an immutable object is that the operation is performed, and an object holding the result is created; and then the target object reference is re-bound to refer to the result object rather than the object it referred to before.

The second subtlety is that `a operator= b` is not quite the same as `a = a operator b`. The augmented version looks `a`'s value only once, so it is potentially faster.

```

>>> name = 'mike'
>>> name + 'chyson'
'mikechyson'
>>> name += ' chyson'
>>> name
'mike chyson'
>>> a = [1, 2, 3]

>>> a + [4]
[1, 2, 3, 4]
>>> a += [4]
>>> a
[1, 2, 3, 4]
```

Python overloads the + and += operators for both strings and lists, the former mearning concatenation and the latter meaning append for strings and extend (append another list) for lists.

1.6 Input/output

Rediction:

- > (output)
- < (input)

Function:

- input()
- print()

1.7 Creating and calling functions

```
def function_name(arguments):
    suite
```

The `arguments` are optional and multiple arguments must be comma-separated. Every Python function has a return value; this defaults to `None` unless we return from the function using the syntax `return value`, in which case `value` is returned.

`def` is a statement that works in a similar way to the assignment operator. When `def` is executed a function object is created and an object reference with the specified name is created and set to refer to the function object. Since functions are objects, they can be stored in collection data types and passed as arguments to other functions.

Although creating our own functions can be very satisfying, in many cases it is not necessary. This is because Python has a lot of functions built in, and a great many more functions in the modules in its standard library, so what we want may well already be available.

A Python module is just a `.py` file that contains Python code. To access the functionality in a module we must import is. For example:

```
import sys
```

To import a module we use the `import` statement followed by the name of the `.py` file, but omitting the extension. Once a module has been imported, we can access any functions, classes, or variables that it contains. For example:

```
print(sys.argv)
```

In general, the syntax for using a function from a module is `module_name.function_name(arguments)`. It makes use of the dot (access attribute) operator.

It is conventional to put all the import statements at the beginning of `.py` files, after the shebang line, and after the modules documentation. We recommend importing standard library modules first, then third-party library modules, and finally your own modules.

Chapter 2

Data types

2.1 Identifiers and keywords

The name we give to our object references are called **identifiers** or just plain **names**.

A valid Python identifier is a **nonempty** sequence of characters of any length that consists of a “start character” and zero or more “continuation characters”. Such an identifier must obey a couple of rules and ought to follow certain conventions.

Rules:

- The start character can be anything that Unicode considers to be a letter.
- Each continuation character can be any character that is permitted as a start character, or any nonwhitespace character.
- No identifier can have the same name as one of Python’s keywords.

Python has a built-in function called `dir()` that returns a list of object’s attributes.

Conventions:

- Don’t use the name of any of Python’s predefined identifiers for your own identifiers.
- Names that begin and end with two underscores should not be used.

2.2 Integral types

Python provides two built-in integral types, `int` and `bool`. Both integers and Boolean are imuutable. When used in Boolean expressions, 0 and `False` are `False` and any other integer and `True` are `True`. When used in numerical expressions `True` evaluates to 1 and `False` to 0.

2.2.1 Integers

Integer literals are written using base 10 by default, but other number bases can be used.

```

>>> 14600926                          # decimal
14600926
>>> 0b110111101100101011011110        # binary
14600926
>>> 0o67545336                        # octal
14600926
>>> 0xDECADE                          # hexadecimal
14600926
```

Binary numbers are written with a leading `0b`, octal numbers with a leading `0o`, and hexadecimal numbers with a leading `0x`. Uppercase letters can also be used.

All the usual mathematical functions and operators can be used with integers. Some of the functionality is provided by functions and other functionality is provided by `int` operators.

Provided by operators:

x + y

x - y

x * y

x / y Divdes x by y; always produces a `float` (or a `complex` if x or y is `complex`)

x // y Divdes x by y; truncates any fractional part so always produces an `int` result

x % y

x ** y Same to x^y

-x Negates x;

+x Does nothing; is sometimes used to clarify code

x | y Bitwise `OR`

x ^ y Bitwise `XOR`

x & y| Bitwise `AND`

x >> y Shifts `i` left by `j`; like `i * (2 ** j)` without overflow checking

x >> y Shifts `i` right by `j`; like `i // (2 ** j)` without overflow checking

~i Inverts `i`'s bits

Provided by functions:

abs(x) Return the absolute value of x

divmod(x, y) Return the quotient and remaninder of dividing x by y as a tuple of two `ints`

pow(x, y) Same to x^y

pow(x, y, z) A faster alternative to $(x**y)\%z$

round(x, n) Return x rounded to n integral digits if n is a negative `int` or return x rounded to n decimal places if n is a positive `int`; the returned value has the same type as x;

bin(i) Return the binary representation of `int i` as a string

hex(i)

int(x) Convert object x to an integer

int(s, base) Convert `str s` to an integer. `base` should be an integer between 2 and 36 inclusive.

oct(i)

All the binary numeric operators (+, -, /, //, %, and **) have augmented assignment versions (+=, -=, /=, //=, %=, and **=) where `x op= y` is logically equivalent to `x = x op y` in the normal case when reading xs value has no side effects.

All the binary bitwise operators (|, ^, &, <<, and >>) have augmented assignment versions (|=, ^=, &=, <<=, and >>=) where `i op= j` is logically equivalent to `i = i op j` in the normal case when reading is value has no side effects.

When an object is created using its data types there are 3 possible use cases:

1. When a data type is called with no arguments an object with a default value is created.
2. When the data type is called with a sinle argument, if an argument of the same type is given, a new object which is a shallow copy of the original object is created. If argument of a different type is given, a conversion is attempted.
3. If two or more arguments are given — not all types support this, and for those that do the argument types and their meanings vary.

2.2.2 Booleans

There are two built-in Boolean objects: `True` and `False`.

2.3 Floating-point types

Python provides three kinds of floating-point values:

- the built-in `float`
- the built-in `complex`
- the `decimal.Decimal` type from the standard library

All three are immutable.

`decimal.Decimal` perform calculations that are accurate to the level of precision we specify (by default, to 28 decimal places) and can represent periodic numbers like 0.1 exactly; but processing is a lot slower than with `float`s. Because of their accuracy, `decimal.Decimal` numbers are suitable for financial calculations.

2.3.1 Floating-point numbers

```

import sys

# the samllest difference that the machine can distinguish
# between two floating-point numbers
sys.float_info.epsilon

help(sys.float_info)
```

The `math` module provides many functions that operate on `float`s. The `math` module is very dependent on the underlying math library that Python was compiled against. This means that some error conditions and boundary cases may behave differently on different platforms.

math.copysign(x, y) Returns `x` with `y`'s sign

math.e

math.pi

math.exp(x)

math.factorial(x) Returns $x!$

math.floor(x)

math.ceil(x)

math.hypot(x, y) Returns $\sqrt{(}x^2 + y^2)$

math.isinf(x) Returns `True` if `float x` is $\pm$inf($\pm\infty$)

math.isnan(x) Returns `True` if `float x` is nan ("not a number")

math.log(x, b) Returns $\log_b x$

math.log10(x) Returns $\log_{10} x$

math.sqrt(x)

2.3.2 Complex numbers

The `complex` data type is an immutable type that holds a pair of `float`s, one representing the real part and the other the imaginary part of a complex number.

```

>>> z = 1.0 + 2.0j
>>> z.real, z.imag
(1.0, 2.0)
```

The functions in the `math` module do not work with complex numbers while `cmath` module does.

2.3.3 Decimal numbers

```

>>> import decimal
>>> a = decimal.Decimal(1234)
>>> b = decimal.Decimal('54321.012345678987654321')
>>> a + b
Decimal('55555.012345678987654321')
```

Numbers of type `decimal.Decimal` work within the scope of a **context**; the context is a collection of settings that affect how `decimal.Decimal` behave.

```

>>> 23 / 1.05
21.904761904761905
>>> print(23 / 1.05)
21.904761904761905
>>> print(decimal.Decimal(23) / decimal.Decimal(1.05))
21.90476190476190383546015179
>>> decimal.Decimal(23) / decimal.Decimal(1.05)
Decimal('21.90476190476190383546015179')
```

When we call `print()` on the result of
`decimal.Decimal(23) / decimal.Decimal(1.05)` the bare number is printed — this output is in **string form**. If we simply enter the expression we get a `decimal.Decimal` output — this output is in **representational form**. All Python objects have two output forms. String form is designed to be human-readable. Representational form is designed to produce output that if fed to a Python interpreter would (when possible) reproduce the represented object.

2.4 Strings

Strings are represented by the immutable `str` data type which holds a sequence of Unicode characters.

triple quoted string:

```

text = """hello world"""
```

Python uses newline as its **statement terminator**, except inside parentheses (()), square brackets ([]), braces (), or triple quoted strings.

All of Python's escape sequences are shown in Table 2.1.

Escape	Meaning
newline	Escape (i.e., ignore) the newline
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII bell (BEL)
\b	ASCII backspace (BS)
\f	ASCII formfeed (FF)
\n	ASCII linefeed (LF)
\N{*name*}	Unicode character with the given name
ooo	Character with the given octal value
\r	ASCII carriage return (CR)
\t	ASCII tab (TAB)
\u*hhhh*	Unicode character with the given 16-bit hexadecimal value
\U*hhhhhhhh*	Unicode character with the given 32-bit hexadecimal value
\v	ASCII vertical tab (VT)
\x*hh*	Character with the given 8-bit hexadecimal value

Figure 2.1: Python's string escapes

In some situations — for example, when writing regular expressions — we need to create strings with lots of literal backslashes. This can be inconvenient since each one must be escaped:

```

import re
phone1 = re.compile("^((?:[(]\\d+[)])?\\s*\\d+(?:-\\d+)?)$")
```

The solution is to use **raw** strings. These are quoted or triple quoted strings whose first quote is preceded by the letter `r`. Inside such strings all characters are taken to be literals, so no escaping is necessary.

```

phone2 = re.compile(r"^((?:[(]\d+[)])?\s*\d+(?:−\d+)?)$")
```

```

>>>'\N{euro sign}'
''
```

If we want to know the Unicode code point for a particular character in a string, we can use the built-in `ord()` function:

```

>>> ord('')
8364
>>> hex(ord(''))
'0x20ac'
>>> '\u20ac'
''
```

we can convert any integer that represents a valid code point into the corresponding Unicode character using the built-in `chr()` function:

```

>>> chr(8734)
''
>>> chr(8364)
''
>>> ascii('')
"'\\u20ac'"
```

2.4.1 Comparing strings

Strings support the usual comparison operators <, <=, ==, !=, >, and >=. These operators compare strings byte by byte in memory.

2.4.2 Slicing and striding strings

```

s = "Light ray"
```

Figure 2.2 shows all the valid index postions for string `s`.

The slice operator has three syntaxes:

`seq[start]`
`seq[start:end]`
`seq[start:end:step]`

Using `+` to concatenate and `+=` to append is not particularly efficient when String many strings are involved. For joining lots of strings it is usually best to use the `str.join()` method.

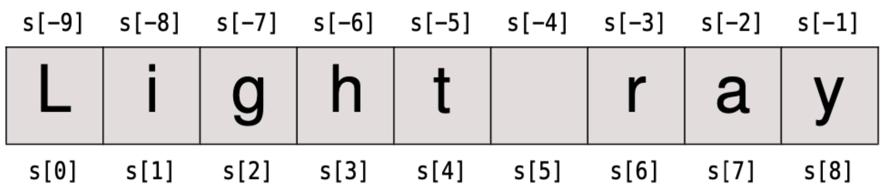

Figure 2.2: String index position

2.4.3 String operators and methods

Since strings are immutable sequences, all the functionality that can be used with imuutable sequences can be used with strings.

- membership (in)
- concatenation (+)
- appedning (+=)
- replication (*)
- augmented assignment replication (*=)

There are some common string methods:

s.capitalize()

s.lower()

s.title()

s.upper()

s.swapcase()

s.center(width, char)

s.ljust(width, char)

s.rjust(width, char)

s.count(t, start, end)

s.encode(encoding, err)

s.startswith(x, start, end)

s.endswith(x, start, end)

s.expandtabs(size)

s.find(t, start, end)

s.index(t, start, end)

s.format(...)

s.isalnum()

s.isalpha()

s.isdecimal()

s.isdigit()

s.isidentifier()

s.islower()

s.istitle()

s.isupper()

s.isnumeric()

s.isprintable()

s.isspace()

s.join(seq)

s.partition(t)

s.replace(t, u, n)

s.split(t, n)

s.splitlines(f)

s.strip(chars)

s.maketrans()

s.translate()

s.zfill(w)

2.4.4 String formatting with the str.format() method

The `str.format()` method returns a new string with the replacement fields in its string replaced with its arguments suitablely formatted.

```
>>> "{0} {1} {2}".format("Hello", 'world', 'mike')
'Hello world mike'
```

Each replacement field is identified by a field name in braces. If the field name is a simple integer, it is taken to be the index position of one of the arguments passed to `str.format()`.

If we need to include braces inside format strings, we can do so by doubling them up.

```
>>> 'just {{{0}}}'.format('brace')
'just {brace}'
```

The replacement field can have any of the following general syntaxes:

```
{field_name}
{field_name!conversion}
{field_name:format_specification}
{field_name!conversion:format_specification}
```

2.4.5 Field names

A field name can be either an integer corresponding to one of the `str.format()` methods arguments, or the name of one of the methods keyword arguments.

```
>>> '{who} solve a leetcode problem every {0} days'.format(1, who='Mike')
'Mike solve a leetcode problem every 1 days'
```

Notice that in an argument list, keyword arguments always come after positional arguments.

If the arguments are collections data types like lists or dictionaries, or have attributes, we can access the part using [] or . notation.

```
>>> stock = ["paper", "envelopes", "notepads"]
>>> "We have {0[1]} and {0[2]} in stock".format(stock)
'We have envelopes and notepads in stock'

>>> d = dict(animal="elephant", weight=12000)
>>> "The {0[animal]} weighs {0[weight]}kg".format(d)
'The elephant weighs 12000kg'

>>> "math.pi=={0.pi}".format(math)
'math.pi==3.14159265359'
```

The local variables that are currently in scope are available from the built-in `locals()` function. This function returns a dictionary whose keys are local variable names and whose values are references to the variables' values. We can use **mapping unpacking** to feed this dictionary into the `str.format()` method. The mapping unpacking operator is `**` and it can be applied to a mapping (such as dictionary) to produce a key-value list suitable for passing to a function. For example:

```

>>> element = "Silver"
>>> number = 47
>>> "Element {number} is {element}".format(**locals())
'Element 47 is Silver'
```

2.4.6 conversions

```

>>> decimal.Decimal("3.4084")
Decimal('3.4084')
>>> print(decimal.Decimal("3.4084"))
3.4084
```

The first is in representational form. The purpose of this form is to provide a string which if interpreted by Python would re-create the ojbect it represents. Not all objects can provide a reproducing representation, in which case they provide a string enclosed in angle brackets. For example `"module 'sys' (built-in)>"`.

The second is in its string form. This form is aimed at human readers, so the concern is to show something that makes sense to people. If a data type doesnt have a string form and a string is required, Python will use the representational form.

Pythons built-in data types know about `str.format()`, and when passed as an argument to this method they return a suitable string to display themselves. In addition, it is possible to override the data types normal behavior and force it to provide either its string or its representational form. This is done by adding a conversion specifier to the field. Currently there are three such specifiers:

- `s` to force string form,
- `r` to force representational for
- `a` to for representational form but only using ASCII characters.

```

>>> "{0} {0!s} {0!r} {0!a} {1!r} {1!a}".format(decimal.Decimal(1), "")
"1 1 Decimal('1') Decimal('1') '' '\\u4f60\\u597d'"
```

2.4.7 Format specifications

Specification for string

For strings, the things that we can control are:

- the fill character,
- the alignment within the field, and
- the minimum and
- maximum field widths.

```
: fill     align        min_width   .max_width
            < for left
            > for right
            ^ for center
```

```

>>> s = "The sword of truth"
>>> '{0}'.format(s)
'The sword of truth'
>>> '{0:25}'.format(s)
'The sword of truth       '
>>> '{0:>25}'.format(s)
'       The sword of truth'
>>> '{0:->25}'.format(s)
'--------The sword of truth'
>>> '{0:.<25}'.format(s)     # the left alignment can not be omitted
'The sword of truth.......'
>>> '{0:.10}'.format(s)
'The sword '
```

Specification for integer

For integers, the format specification allows us to control:

- the fill character,
- the alignment within the field,
- the sign,
- whether to use a nonlocale-aware comma separator to group digits,
- the minimum field width, and
- the number base.

```
: fill  alignment      sign              #      width   ,         type
        = pad between  + force sign;   prifix           use       b,c,d
        sign and       - sign if       ints             commas    n,o,x,
        digits         needed;         with             for       X
        for numbers    " " space or    0b, 0o,          grouping
                       - as            or 0x
                       appropriate
```

```
>>> '{0:0=12}'.format(-1234)
'-00000001234'

>>> "[{0: }] [{1: }]".format(539802, -539802) # space or - sign
'[ 539802] [-539802]'
>>> "[{0:+}] [{1:+}]".format(539802, -539802) # force sign
'[+539802] [-539802]'
>>> "[{0:-}] [{1:-}]".format(539802, -539802) # - sign if needed
'[539802] [-539802]'

>>> "{0:b} {0:o} {0:x} {0:X}".format(123)
'1111011 173 7b 7B'
>>> "{0:#b} {0:#o} {0:#x} {0:#X}".format(123)
'0b1111011 0o173 0x7b 0X7B'

>>> '{0:,}'.format(1234567890)
'1,234,567,890'
```

The last format character `n` has the same effect as d when given an integer. What makes `n` special is that it respects the current locale and will use locale-specific decimal separator and grouping separator in the output it produces. The default locale is called the C locale, and for this the decimal and grouping characters are a period and an empty string.

```
>>> import locale
>>> x = 1234567890
>>> locale.setlocale(locale.LC_ALL, 'C')
'C'
>>> '{:n}'.format(x)
'1234567890'
>>> locale.setlocale(locale.LC_ALL, 'en_US.UTF-8')
'en_US.UTF-8'
>>> '{:n}'.format(x)
'1,234,567,890'
>>> locale.setlocale(locale.LC_ALL, 'de_DE.UTF-8')
'de_DE.UTF-8'
>>> '{:n}'.format(x)
'1234567890'
```

Specification for floating

For floating-point numbers, the format specification gives us control over:

- the fill character,

- the alignment within the field,
- the sign,
- whether to use a non-locale aware comma separator to group digits,
- the minimum field width,
- the number of digits after the decimal place, and
- whether to present the number in standard or exponential form, or as a percentage.

```
: fill  alignment  sign  width  ,  .precision      type
                                   number of       e,E,f,
                                   decimal places  g,G,n,
                                                   %
```

- e for exponential form with lowercase e
- E for exponential form with lowercase E
- f for standard floating-point form
- g for “general” formthis is the same as f unless the number is very large, in which case it is the same as e
- G is is almost the same as g, but uses either f or E
- % for percentage

```

>>> '{0:12.2e}'.format(math.pi)
'    3.14e+00'
>>> '{0:12.2f}'.format(math.pi)
'        3.14'
>>> '{:,.6f}'.format(1234567890.1234567890)
'1,234,567,890.123457'
>>> "{:,.4f}".format(3.59284e6-8.984327843e6j)
'3,592,840.0000-8,984,327.8430j'
```

Character encodings

Unicode assigns every character to an integer — called a **code point** in Unicode-speak. Nowadays, Unicode is usually stored both on disk and in memory using UTF-8, UTF-16, or UTF-32. The first of these, UTF-8, is backward compatible with 7-bit ASCII since its first 128 code points are represented by single-byte values that are the same as the 7-bit ASCII character values. To represent all the other Unicode characters, UTF-8 uses two, three, or more bytes per character.

A lot of other software, such as Java, uses UCS-2 (which in modern form is the same as UTF-16). This representation uses two or four bytes per character, with the most common characters represented by two bytes. The UTF-32 representation (also called UCS-4) uses four bytes per character. Using UTF-16 or UTF-32 for storing Unicode in files or for sending over a network connection has a potential pitfall: If the data is sent as integers then the endianness matters. One solution to this is to precede the data with a byte order mark so that readers can adapt accordingly. This problem doesnt arise with UTF-8, which is another reason why it is so popular.

Python represents Unicode using either UCS-2 (UTF-16) format, or UCS-4 (UTF-32) format. In fact, when using UCS-2, Python uses a slightly simplified version that always uses two bytes per character and so can only represent code points up to 0xFFFF. When using UCS-4, Python can represent all the Unicode code points. The maximum code point is stored in the read-only sys.maxunicode attributeif its value is 65535, then Python was compiled to use UCS-2; if larger, then Python is using UCS-4.

Chapter 3

Collection data types

3.1 Sequence types

A **sequence** type is one that support:

- the membership operator (`in`)
- the size function (`len()`)
- slices ([])
- and is iterable.

Python provides five built-in sequence types:

- bytearray
- bytes
- list
- str
- tuple

When iterated, all of these sequences provide their items in order.

3.1.1 Tuples

Tuples are immutable. Tuples are able to hold any items of any data type, including collection types such as tuples and lists, since what they really hold are object references.

Tuples provide just two methods, `t.count(x)` and `t.index(x)`.

tuple coding style: omit parentheses:

- tuples on the left-hand size of a binary operator
- on the right-hand size of a unary statement

other cases with parentheses.

```
a, b = (1, 2)  # left of binary operator
del a, b       # right of unary operator
```

When we have a sequences on the right-hand side of an assignment, and we have a tuple on the left-hand side, we say that the right-hand side has been **unpacked**. Sequence unpacking can be used to swap values, for example:

```
a, b = (b, a) # or a, b = b, a
# the parentheses here are for code style

for x, y in ((3, 4), (5, 12), (28, -45)):
    print(math.hypot(x, y))
```

3.1.2 Named tuples

A named tuple behaves just like a plain tuple, and has the same performance characteristics. What it adds is the ability to refer to items in the tuple by name as well as by index position.

```
import collections

Fullname = collections.namedtuple('Fullname',
                                  'firstname middlename lastname')
persons = []
persons.append(Fullname('Mike', 'Ming', 'Chyson'))
persons.append(Fullname('Alfred', 'Bernhard', 'Nobel'))
for person in persons:
    print('{firstname} {middlename} {lastname}'.format(**person._asdict()))
```

3.1.3 Lists

List are mutable. Since all the items in a list are really object references, lists can hold items of any data type, including collection types such as lists and tuples.

Although we can use the slice operator to access items in a list, in some situations we want to take two or more pieces of a list in one go. This can be done by sequence unpacking. Any iterable (lists, tuples, etc.) can be unpacked using the sequence unpacking operator, an asterisk or star (*). When used with two or more variables on the left-hand side of an assignment, one of which is preceded by *, items are assigned to the variables, with all those left over assigned to the starred variable. Here are some examples:

```

>>> a = list(range(10))
>>> first, *last = a
>>> print(first, last)
0 [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> first, *middle, last = a
>>> print(first, middle, last)
0 [1, 2, 3, 4, 5, 6, 7, 8] 9
>>>
>>> *first, last = a
>>> print(first, last)
[0, 1, 2, 3, 4, 5, 6, 7, 8] 9
```

List methods:

list.append(x)

list.count(x)

list.extend(m)

list += m

list.index(x, start, end)

list.insert(i, x)

list.pop() Returns and removes the rightmost item of list

list.pop(i)

list.remove(x) Removes the leftmost occurrence of item x from list

list.reverse() Reverses list in-place

list.sort(...) Sorts list in-place

Individual items can be replaced in a list by assigning to a particular index position. Entire slices can be replaced by assigning an iterable to a slice. The slice and the iterable don't have to be the same length. In all cases, the slice's items are removed the the iterable's items are inserted.

```

>>> nums = [0, 1, 2, 3, 4, 5, 6]
>>> nums[0] = 100
>>> nums
[100, 1, 2, 3, 4, 5, 6]
>>> nums[2:2] = [200]    # same to nums.insert(2, 200)
>>> nums
[100, 1, 200, 2, 3, 4, 5, 6]
>>> nums[2:4] = [10, 11, 12, 13, 14]
>>> nums
[100, 1, 10, 11, 12, 13, 14, 3, 4, 5, 6]
```

In lists, striding allows us to access every n-th item which can often be useful. For example:

```

>>> x = list(range(1, 11))
>>> x
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> x[1::2] = [0] * len(x[1::2])
>>> x
[1, 0, 3, 0, 5, 0, 7, 0, 9, 0]
```

```
>>> x = list(range(-5, 5))
>>> x
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> x.sort(key=lambda x: x**2)
>>> x
[0, -1, 1, -2, 2, -3, 3, -4, 4, -5]
```

For inserting items, lists perform best when items are added or removed at the end (`list.append(), list.pop()`). The worst performance occurs when we search for items in a list, for example, using `list.remove()` or `list.index()`, or using `in` for membership testing. If fast searching or membership testing is required, a `set` or a `dict` may be a more suitable collection choice. Alternatively, lists can provide fast searching if they are kept in order by sorting them and using a binary search (provided by the `bisect` module), to find items.

3.1.4 List comprehensions

A **list comprehension** is an expression and a loop with optional condition enclosed in brackets where the loop is used to generate items for the list, and where the condition can filter out unwanted items.

```
[expression for item in iterable]
[expression for item in iterable if condition]
```

```
leaps = [y for y in range(1900, 1940)
         if (y % 4) == 0 and y % 100 != 0) or (y % 400 == 0)]
```

If the generated list is very large, it may be more efficient to generate each item as it is needed rather than produce the whole list at once. This can be achieved by using a generator rather than a list comprehension.

3.2 Set types

A **set** is a collection data type that supports:

- the membership operator(`in`),
- the size function (`len()`),
- and is iterable.

Python provides two built-in set types:

- the mutable `set` type
- the immutable `frozenset`

When iterated, set types provide their items in an arbitrary order.

Only **hashable** objects may be added to a set. Hashable objects are objects which have a `__hash__()` special method whose return value is always the same thoughout the ojbect's lifetime, and which can be compared for equality using the `__eq__()` special method. (Special mehtods are methods whose name begins and ends with two underscores)

3.2.1 Sets

A **set** is an unordered collection of zero or more object references that refer to hashable objects. Sets are mutable. Sets always contain unique items — adding duplicate items is safe but pointless.

Set methods:

s.add(x)

s.clear()

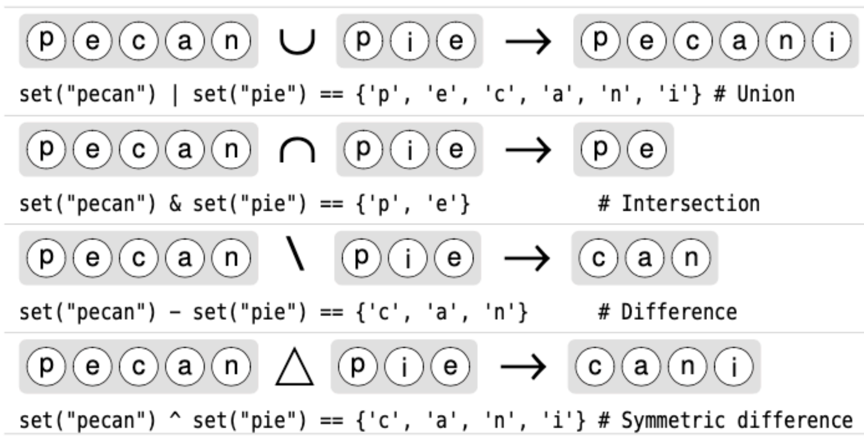

Figure 3.1: Standard set operators

s.copy()

s.difference(t) Same to s - t

s.difference_update(t) Same to s -= t

s.discard Removes item x from set s if it is in s

s.remove() Removes item x from set s, or raises a KeyError exception if x is not in s

s.intersection(t) Same to s & t

s.intersection_update(t) Same to s &= t

s.isdisjoint(t) Returns True if sets s and t have no items in common

s.issubset(t) Same to s <= t

s.issuperset(t) Same to s >= t

s.pop() Returns and remove a random item from set s, or raises a KeyError exception if s is empty

s.symmetric_difference(t) Same to s ˆ t

s.symmetric_difference_update(t) Same to sˆ= t

s.union(t) Same to s | t

Sets are used used for fast membership test and removring duplicated items.

3.2.2 Set comprehensions

```
{expression for item in iterable}
{expression for item in iterable if condition}
```

3.2.3 Frozen sets

A frozen set is a set that, once created, cannot be changed. Since frozen set are immutable, sets and frozen sets can contain frozen sets.

3.3 Mapping types

A **mapping** type is one that supports:

- the membership operator (in)
- the size function (len())
- is iterable

Mappings are collection of key-value items and provide methods for accessing items and their keys and values.

When iterated, unordered mapping types provide their items in an arbitrary order.

There are one built-in mapping types and two standard library's mapping types:

- dict
- collections.defaultdict
- collections.OrderedDict

3.3.1 Dictionaries

A `dict` is an unordered collection of zero or more keyvalue pairs whose keys are object references that refer to hashable objects, and whose values are object references referring to objects of any type. Dictionaries are mutable.

```
>>> d = dict()
>>> d
{}
>>> d = {}
>>> d
{}
>>> d = {"hello": 1, "world": 2}
>>> d
{'hello': 1, 'world': 2}
>>> d = dict(hello=1, world=2)
>>> d
{'hello': 1, 'world': 2}
>>> d = dict([("hello", 1), ("world", 2)])
>>> d
{'hello': 1, 'world': 2}
```

Dictionary methods:

d.clear()

d.copy()

d.fromkeys(s, v) Returns a dict whose keys are the items in sequence s and whose values are None or v if v is given

d.get(k) Returns key k's associated value or None if k isn't in dict d

d.get(k, v) Returns key k's associated value, or v if k isn't in dict d

d.items()

d.keys()

d.values()

d.pop(k)

d.pop(k, v)

d.popitem() Returns and removes an arbitrary (key, value) pair from dict d

d.setdefault(k, v) The same as the dict.get() method, except that if the key is not indict d,a new item is inserted with the key k, and with a value of None or of v if v is given

d.update(a) Adds every (key, value) pair from a that isnt in dict d to d, and for every key that is in both d and a, replaces the corresponding value in d with the one in a – a can be a dictionary, an iterable of (key, value) pairs, or keyword arguments

The `dict.items()`, `dict.keys()`, and `dict.values()` methods all return dictionary views. A dictionary view is effectively a read-only iterable object that appears to hold the dictionarys items or keys or values, depending on the view we have asked for.

In general, we can simply treat views as iterables. However, two things make a view different from a normal iterable. One is that if the dictionary the view refers to is changed, the view reflects the change. The other is that key and item views support some set-like operations. Given dictionary view `v` and `set` or dictionary view `x`, the supported operations are:

```
v & x  # intersection
v | x  # union
v - x  # difference
v ^ x  # symmetric difference
```

```
>>> d = {}.fromkeys("abcd", 3)
>>> d
{'a': 3, 'b': 3, 'c': 3, 'd': 3}
>>> s = set("abc")
>>> s
{'a', 'c', 'b'}
>>> d.keys() & s
{'a', 'c', 'b'}
>>>
>>> d
{'a': 3, 'b': 3, 'c': 3, 'd': 3}
>>> d.setdefault('a')
3
>>> d
{'a': 3, 'b': 3, 'c': 3, 'd': 3}
>>> d.setdefault('z', 100)
100
>>> d
{'a': 3, 'b': 3, 'c': 3, 'd': 3, 'z': 100}
```

3.3.2 Dictionary comprehensions

```
{keyexpression: valueexpression for key, value in iterable}
{keyexpression: valueexpression for key, value in iterable if condition}
```

```
import os

# filename: filesize
d = {name: os.path.getsize(name) for name in os.listdir('.') if os.path.isfile(name)}
print(d)
```

```

# revert dict
inserted_d = {v: k for k, v in d.items()}
print(inserted_d)
```

3.3.3 Default dictionaries

Default dictionaries are dictionaries — they have all the operators and methods that dictionaries provide. What makes default dictionaries different from plain dictionaries is the way they handle missing keys.

When a default dictionary is created, we can pass in a **factory function**. A factory function is a function that, when called, returns an object of a particular type. All of Pythons built-in data types can be used as factory functions. The factory function passed to a default dictionary is used to create default values for missing keys.

> Note that the **name** of a function is an object reference to the function — so when we want to pass functions as parameters, we just pass the name. When we use a function with parentheses, the parentheses tell Python that the function should be called.

```
>>> words = collections.defaultdict(int)
>>> words
defaultdict(<class 'int'>, {})
>>> words['hello'] += 1
>>> words
defaultdict(<class 'int'>, {'hello': 1})
>>> words['hello'] += 1
>>> words
defaultdict(<class 'int'>, {'hello': 2})
>>>
>>> de = collections.defaultdict(lambda : "Thanks to ")
>>> de
defaultdict(<function <lambda> at 0x7fa999a75a60>, {})
>>> de['Mike'] += 'Mike'
>>> de
defaultdict(<function <lambda> at 0x7fa999a75a60>, {'Mike': 'Thanks to Mike'})
```

3.3.4 Ordered dictionaries

The ordered dictionaries type is `collections.OrderedDict` Ordered dictionaries store their items in the order in which they were inserted. If we change an item's value, the order is not changed.

```
d = collections.OrderedDict([('z', -4), ('e', 19), ('k', 7)])
for k in d:
    print(k, d[k])

tasks = collections.OrderedDict()
```

```
tasks[8031] = "Backup"
tasks[4027] = "Scan Email"
tasks[5733] = "Build System"
for k in tasks:
    print(k, tasks[k])
```

If we want to move an item to the end, we must delete it and then reinsert it. We can also call `popitem()` to remove and return the last keyvalue item in the ordered dictionary; or we can call `popitem(last=False)`, in which case the first item will be removed and returned.

3.4 Iterating and copying collections

3.4.1 Iterators and iterable operations and functions

An **iterable** data type is one that can return each of its items one at a time. Any object that has an `__iter__()` method, or any sequence (i.e. an object that has a `__getitem__()` method taking integer arguments starting from 0) is an iterable and can be provide an **iterator**. An iterator is an object that provides a `__next__()` method which returns each successive item in turn, and raises a `StopIteration` exception when there are no more items.

The operators and functions that can be used with iterables:

s + t Returns a sequence that is the concatenation of sequences s and t

s * t Returns a sequences that is int n concatenation of sequences s

x in i Returns True if item x is in iterable i

all(i) Returns True if every item in iterable i evalueates to True

any(i) Returns True if any item in iterable i evalueates to True

enumerate(i, start) Normally used in for ... in loops to provide a sequence of (index, item) tuples with indexes starting at 0 or start

len(x)

max(i, key) Returns the biggest item in iterable i or the item with the biggest key(item) value if a key function is given

min(i, key) Returns the smallest item in iterable i or the item with the smallest key(item) value if a key function is given

range(start, stop, step) Returns an integer iterator.

reversed(i) Returns an iterator that returns the items from iterator i in reverse order

sorted(i, key, reverse) Return a list of the items from iterator i in sorted order; key is used to provide DSU (Decorate, Sort, Undecorate) sorting. If reverse is True the sorting is done in reverse order.

sum(i, start) Returns the sum of the items in iterable i plus start (which defaults to 0)

zip(i1, ..., iN) Returns an iterator of tuples using the iterators i1 to iN

The order in which items are returned depends on the underlying iterable. In the case of lists and tuples, items are normally returned in sequential order starting from the first item (index position 0), but some iterators return the items in an arbitrary order — for example, dictionary and set iterators.

The built-in `iter()` function has two quite different behaviors.

- When given a collection data type or a sequence it returns an iterator for the oject it is passed — or raise a `TypeError` if the object cannot be iterable.

- When given a callable (a function or method) and a sentinel value, the function passed in is called once at each iteration, returning teh function'sreturn value each time, or raising a `StopIteration` exception if the return value equals the sentinel.

When we use a `for item in iterable` loop, Python in effect calls `iter(iterable)` to get an iterator. This iterator's `__next__()` method is then called at each loop iteration to get the next item, and when the `StopIteration` exception is raised, it is caughted and the loop is terminated.

```
# manner 1
product = 1
for i in [1, 2, 4, 8]:
    product *= i
print(product)


# manner 2
product = 1
i = iter([1, 2, 4, 8])
while True:
    try:
        product *= i
    except StopIteration:
        break
print(product)
```

Any (finite) iterable, i, can be converted into a tuple by calling `tuple(i)`, or can be converted into a list by calling `list(i)`.

```
>>> x = []
>>> for t in zip(range(-10, 0, 1), range(0, 10, 2), range(1, 10, 2)):
...     x += t
...
>>> x
[-10, 0, 1, -9, 2, 3, -8, 4, 5, -7, 6, 7, -6, 8, 9]
>>> sorted(x)
[-10, -9, -8, -7, -6, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sorted(x, reverse=True)
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -6, -7, -8, -9, -10]
>>> sorted(x, key=abs)
[0, 1, 2, 3, 4, 5, 6, -6, -7, 7, -8, 8, -9, 9, -10]
```

A function's name is an object reference to the function; it is the parentheses that follow the name that tell Python to call the function.

Python's sort algorithm is an adaptive stable mergesort that is both fast and smart, and it is especially well optimized for partially sorted lists. The "adaptive" part means that the sort algorithm adapts to circumstances — for example, taking advantage of partially sorted data. The "stable" part means that the items that sort equally are not moved in relation to each other. When sorting collections of itegers, strings, or other simple types their "less than" operator (<) is used. Python can sort collections that contain collections, working recursively to any depth.

Lists can be sorted in-place using the `list.sort()` method, which takes the same optinal arguments as `sorted()`.

3.4.2 Copying collections

Since Python uses **object references**, when we use the assignment operator (+), no copying takes place. If the right-hand operand is a literal such as a string or a number, the left-hand operand is set to be an object reference that refers to the in-memory object that holds the literals value. If the right-hand operand is an object reference, the left-hand operand is set to be an object reference that refers to the same object as the right-hand operand. One consequence of this is that assignment is very efficient.

For sequences, when we take a slice, the slice is always an independent copy of the items copied. For dictionaries and sets, copying can be achived using `dict.copy()` and `set.copy()`. In addition, the `copy` module provides the `copy.copy()` function that returns a copy of the object it is given. Another way to copy the built-in collection types is to use the type as a function with the collection to be copied as its argument.

Note, thought, that all of these copying techniques are **shallow** — that is, only object references are copied and not the object themselves. For immutable data types like numbers and strings this has the same effect as copying, but for mutable data types such as nested collections this means that the object they refer to are referred to both by the original collection and by the copied collection.

```
print('{:.^50}'.format('print(x,y)'))
x = [53, 68, ['A', 'B', 'C']]
y = x[:]
print(x, y, sep='\n')

print('{:.^50}'.format('print(x,y)'))
y[1] = 40
x[2][0] = 'Q'
print(x, y, sep='\n')

"""
....................print(x,y)....................
[53, 68, ['A', 'B', 'C']]
[53, 68, ['A', 'B', 'C']]
....................print(x,y)....................
[53, 68, ['Q', 'B', 'C']]
[53, 40, ['Q', 'B', 'C']]
"""
```

If we really need independent copies of arbitrarily nested collections, we can deep-copy:

```
import copy

x = [53, 68, ['A', 'B', 'C']]
y = copy.deepcopy(x)
y[1] = 40
x[2][0] = 'Q'
print('{:.^50}'.format('print(x,y)'))
print(x, y, sep='\n')

"""
....................print(x,y)....................
[53, 68, ['Q', 'B', 'C']]
[53, 40, ['A', 'B', 'C']]
"""
```

Chapter 4

Control structures and functions

4.1 Control structures

4.1.1 Conditional branching

```
if boolean_expression1:
    suite1
elif boolean_expression2:
    suite2
...
elif boolean_expressionN:
    suiteN
else:
    else_suite
```

conditional expression:

```
expression1 if boolean_expression else expression2
```

One common programming pattern is to set a variable to a default value, and then change the value if necessary.

```
width = 100 + (10 if margin else 10)
print('{} file{}'.format(count if count != 0 else 'no', 's' if count != 1 else ''))
```

4.1.2 Looping

while loops

```
while boolean_expression:
    while_suite
else:
    else_suite
```

As long as the `boolean_expression` is `True`, the `while` block's suite is executed. If the `boolean_expression` is or becomes False, the loop terminates, and if the optional `else` clause is present, its suite is executed. If the loop does not terminate normally, any optional `else` clause's suite is skipped. That is, if the loop is broken out of due to a `break` statement, or a `return` statement, or if an exception is raised, the `else` clause's suite is not executed.

```python
i = 0
while i < 100:
    i += 1
else:
    last = i
print(last)  # 100
```

```python
def list_find(lst, target):
    """
    Find the first target's index or -1 if not find.

    :param lst:
    :param target:
    :return: index of the target if found or -1 if not found
    """
    index = 0
    while index < len(lst):
        if lst[index] == target:
            break
        index += 1
    else:
        index = -1
    return index
```

for loops

```
for expression in iterable:
    for_suite
else:
    else_suite
```

The rule to run `else_suite` is same for while loop.

```
def list_find2(lst, target):
    for index, x in enumerate(lst):
        if x == target:
            break
    else:
        index = -1
    return index
```

4.2 Exception handling

4.2.1 Catching and raising exceptions

```
try:
    try_suite
except exception_group1 as variable1:
    except_suite1
...
except exception_groupN as variableN:
    except_suiteN
else:
    else_suite
finally:
    finally_suite
```

There must be at least one `except` block, but both the `else` and the `finally` blocks are optional. The `else` blocks suite is executed when the `try` blocks suite has finished normally — but it is not executed if an exception occurs. If there is a `finally` block, it is always executed at the end.

Each `except` clauses exception group can be a single exception or a parenthesized tuple of exceptions.

If an exception occurs in the `try` blocks suite, each `except` clause is tried in turn. If the exception matches an exception group, the corresponding suite is executed. To match an exception group, the exception must be of the same type as the (or one of the) exception types listed in the group, or the same type as the (or one of the) groups exception types subclasses.

```
def lst_find(lst, target):
    try:
        index = lst.index(target)
    except ValueError:
        index = -1
    return index
```

Python offers a simpler `try...finally` block:

```
try:
    try_suite
finally:
    finally_suite
```

```
# remove black lines
def read_data(filename):
    lines = []
    fh = None
    try:
        fh = open(filename)
        for line in fh:
            if line.strip():
                lines.append(line)
    except (IOError, OSError) as err:
        print(err)
        return []
    finally:
        if fh is not None:
            fh.close()
    return lines
```

Rasing exceptions

Exceptions provide a useful means of changing the flow of control.

There are three syntaxes for raising exceptions:

```
raise exception(args)
raise exception(args) from original_exception
raise
```

If we give the exception some text as its argument, this text will be output if the exception is printed when it is caught. When the third syntax is used, `raise` will reraise the currently active exception — and if there isn't one it will raise a `TypeError`.

4.2.2 Custom exceptions

Custom exceptions are custom data types (classes).

```
class exceptionName(baseException): pass
```

The base class should be `Exception` or a class that inherits from `Exception`.

One use of custom exceptions is to break out of deeply nested loops.

```python
def find_word(table, target):
    found = False
    for row, record in enumerate(table):
        for column, field in enumerate(record):
            for index, item in enumerate(field):
                if item == target:
                    found = True
                    break
            if found:
                break
        if found:
            break

    if found:
        print('found at ({}, {}, {})'.format(row, column, index))
    else:
        print('not found')


def find_word2(table, target):
    class FoundException(Exception):
        pass

    try:
        for row, record in enumerate(table):
            for column, field in enumerate(record):
                for index, item in enumerate(field):
                    if item == target:
                        raise FoundException
    except FoundException:
        print('found at ({}, {}, {})'.format(row, column, index))
    else:
        print('not found')
```

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
```

```
|    +-- IndexError
|    +-- KeyError
+-- MemoryError
+-- NameError
|    +-- UnboundLocalError
+-- OSError
|    +-- BlockingIOError
|    +-- ChildProcessError
|    +-- ConnectionError
|    |    +-- BrokenPipeError
|    |    +-- ConnectionAbortedError
|    |    +-- ConnectionRefusedError
|    |    +-- ConnectionResetError
|    +-- FileExistsError
|    +-- FileNotFoundError
|    +-- InterruptedError
|    +-- IsADirectoryError
|    +-- NotADirectoryError
|    +-- PermissionError
|    +-- ProcessLookupError
|    +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
|    +-- NotImplementedError
|    +-- RecursionError
+-- SyntaxError
|    +-- IndentationError
|         +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|    +-- UnicodeError
|         +-- UnicodeDecodeError
|         +-- UnicodeEncodeError
|         +-- UnicodeTranslateError
+-- Warning
     +-- DeprecationWarning
```

```
+-- PendingDeprecationWarning
+-- RuntimeWarning
+-- SyntaxWarning
+-- UserWarning
+-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning
+-- ResourceWarning
```

4.3 Costom functions

Functions are a means by which we can package up and parameterize functionality. Four kinds of functions can be created in Python:

- global functions
- local functions
- lambda functions
- methods

Global objects (including functions) are accessible to any code in the same module (i.e., the same .py file) in which the object is created. Global objects can also be accessed from other modules.

Local functions (also called nested functions) are functions that are defined inside other functions. These functions are visible only to the function where they are defined.

Lambda functions are expressions, so they can be created at their point of use; however, they are much more limited than normal functions.

Methods are functions that are associated with a particular data types and can be used only in conjunction with the data type.

The general syntax for creating a (global or local) function is:

```
def function_name(parameters):
    suite
```

```
def my_sum(a, b, c=1):
    return a + b + c
```

```python


print(my_sum(1, 2, 3)) # 6
print(my_sum(1, 2))    # 4
```

`a,b` is called **positional arguments**, because each argument passed is set as the value of the parameter in the corresponding position. `c` is called **keyword arguments**, because each argument is passed by keyword not order.

When default values are given they are created at the time the `def` statement is executed (i.e., when the function is created), not when the function is called. For immutable arguments like numbers and strings this doesnt make any difference, but for mutable arguments a subtle trap is lurking.

```python
def append_if_even(x, lst=[]):
    if x % 2 == 0:
        lst.append(x)
    return lst


def append_if_even2(x, lst=None):
    lst = [] if lst is None else lst
    if x % 2 == 0:
        lst.append(x)
    return lst


for i in range(3):
    result1 = append_if_even(i)
    result2 = append_if_even2(i)
    print(f'{result1=},{i=}')
    print(f'{result2=},{i=}')

# result1=[0],i=0
# result2=[0],i=0
# result1=[0],i=1
# result2=[],i=1
# result1=[0, 2],i=2
# result2=[2],i=2
```

This idiom of having a default of `None` and creating a fresh object should be used for dictionaries, lists, sets, and any other mutable data types that we want to use as default arguments.

4.3.1 Names and docstrings

A few rules of good names:

- Use a naming scheme, and use it consistently. For example:
  - `UPPERCASE` for constants
  - `TitleCase` for classes
  - `camelCase` for GUI functions and methods
  - `lowercase` or `lowercase_with_underscores` for everything else
- For all names, avoid abbreviations, unless they are both standardized and widely used.
- Be proprotional with variable and parameter names: `x` is a perfectly good name for an x-coordinate and `i` is fine for a loop counter, but in general the name should be long enough to be descriptive.
- Functions and methods should have names that say what they do or what they return, but never how they do it — since that might change.

We can add documentation to any function by using a **docstring** — this is simply a string that comes immediately after the `def` line, and before the functions code proper begins.

```
def shorten(text, length=25, indicator="..."):
    """Returns text or a truncated copy with the indicator added

    text is any string; length is the maximum length of the returned
    string (including any indicator); indicator is the string added at
    the end to indicate that the text has been shortened

    >>> shorten("Second Variety")
    'Second Variety'
    >>> shorten("Voices from the Street", 17)
    'Voices from th...'
    >>> shorten("Radio Free Albemuth", 10, "*")
    'Radio Fre*'
    """
    if len(text) > length:
        text = text[:length - len(indicator)] + indicator
    return text
```

It is not unusual for a function or methods documentation to be longer than the function itself. One convention is to make the first line of the docstring a brief one-line description, then have a blank line followed by a full description, and then to reproduce some examples as they would appear if typed in interactively.

4.3.2 Argument and parameter unpacking

We can use sequence unpacking operator(*) to supply positional arguments and or mapping unpacking operator(**) to keyword arguemnts.

```
def my_sum(a, b, c=1):
    return a + b + c

print(my_sum(*[1, 2, 3, 4][:3]))    # 6
print(my_sum(*[1, 2], **{'c': 3}))  # 6
```

We can also use the sequence unpacking operator in a function's parameter list. This is useful when we want to create functions that can take a variable number of positional arguments.

```
def product(*args):
    result = 1
    for arg in args:
        result *= arg
    return result
```

Having the * in front means that inside the function the `args` parameter will be a **tuple** with its itmes set to however many positional arguments are given.

```
def product(*args):
    result = 1
    for arg in args:
        result *= arg
    return result


print(product(*list(range(1, 10))))  # 362880
print(math.factorial(9))             # 362880
```

```
# It is also possible to use * as a parameter in its own right.
# This is used to signify that there can be no positional arguments after the *.
def heron(a, b, c, *, units='square meters'):
    s = (a + b + c) / 2
    area = math.sqrt(s * (s - a) * (s - b) * (s - c))
    return f'{area} {units}'


print(heron(25, 24, 7))
print(heron(41, 9, 40, units="sq. inches"))
print(heron(25, 24, 7, "sq. inches"))   # TypeError
```

We can also use the mapping unpacking operator with parameters. This allows us to create functions that will accept as many keyword arguments as are given.

```
def print_dict(**kwargs):
    for key in sorted(kwargs):
        print(f'{key:10} : {kwargs[key]}')


print_dict(**{str(i): f'{100 * i:3}%' for i in range(10)})

# 0          :   0%
# 1          : 100%
# 2          : 200%
# 3          : 300%
# 4          : 400%
# 5          : 500%
# 6          : 600%
# 7          : 700%
# 8          : 800%
# 9          : 900%
```

```
def print_args(*args, **kwargs):
    for i, arg in enumerate(args):
        print("positional argument {0} = {1}".format(i, arg))
    for key in kwargs:
        print("keyword argument {0} = {1}".format(key, kwargs[key]))


print_args(*list(range(10)), **locals())
```

4.3.3 Accessing variables in the global scope

There are two ways to create a global variable:

- Object defined in `.py` level is global variables.
- variables defined with `global` keyword.

Others are local variables.

```
AUTHOR = 'Mike'  # global


def say_hello():  # global
    global language  # global
    language = 'fr'
    text = 'hello'  # local
    print(text)


class MyException(Exception):  # global
    pass


say_hello()
print(language)
```

4.3.4 Lambda functions

Lambda functions are functions created using the following syntax:

```
lambda parameters: expression
```

The **parameters** are optinal, and if supplied they are normally just comma-separated variable names, that is, positional arguments, although the complement argument syntax supported by `def` statements can be used. The `expression` can not contain **branches** or **loops** (although conditional expressions are allowed), and can not have a `return` (or `yield`) statment. The result of a `lambda` expression is an anonymous function. When a lambda function is called it returns the result of computing the `expression` as its result.

```
lst = list(range(-3, 3))
print(lst)
print(sorted(lst, key=lambda x: x ** 2))
print(sorted(lst, key=lambda key=None: key ** 2))  # seldom used

# [-3, -2, -1, 0, 1, 2]
# [0, -1, 1, -2, 2, -3]
# [0, -1, 1, -2, 2, -3]
```

```
s = lambda x: '' if x == 1 else 's'  # use def instead
print(s(1))  #
print(s(2))  # s

p = lambda key='hello': print(key)  # use def instead
p('world')  # world
```

There are two common usage for lambda functions:

- key function
- default value

```
  sorted(lst, key=lambda x: x ** 2))
  message_dict = collections.defaultdict(lambda: 'No message avaiable')
```

4.3.5 Assertions

Preconditions and postconditions can be specified using `assert` statements:

```
assert boolean_expression, optional_expression
```

If the `boolean_expression` evaluates to `False` an `AssertionError` exception is raised. If the optional `optional_expression` is given, it is used as the argument to the `AssertionError` exception.

```
def product(*args):
    assert all(args), "0 argument"
    result = 1
    for arg in args:
        result *= arg
    return result
```

Note: Assertions are designed for developers, not end-users. Once a program is readly for public release, we can tell Python not to execute `assert` statements. This can be done with:

- -0 option in commandline, python -O program.py
- set the `PYTHONOPTIMIZE` environment variable to O.

We can use -OO option to strip out both `assert` statements and docstrings. However, there is no environment variable for setting this option.