**Compiler & Interpreter:**

Compliers and interpreters are programs that help convert the high level language (Source Code) into machine codes to be understood by the computers. Computer programs are usually written on high level languages. A high level language is one that can be understood by humans. To make it clear, they contain words and phrases from the languages in common use – English or other languages for example. However, computers cannot understand high level languages as we humans do. They can only understand the programs that are developed in binary systems known as a machine code. To start with, a computer program is usually written in high level language described as a source code. These source codes must be converted into machine language and here comes the role of compilers and interpreters.

To start with, a compiler creates the program. It will analyze all the language statements to check if they are correct. If it comes across something incorrect, it will give an error message. If there are no errors spotted, the compiler will convert the source code into machine code. The compiler links the different code files into programs that can be run such as exe. Finally the program runs.  
  
An interpreter creates the program. It neither links the files nor generates machine code. The source statements are executed line by line while executing the program.

**Python** is interpreted language means its not required compile the program explicitly.

In case of using compilers, the program codes are translated into machine code already and hence the time to execute the code is very less. On the negative side, it is not possible to change the program without going back to the source code while working with a compiler.  
  
Interpreters make working with the source code much easier. Hence they are highly suitable especially for the beginners. On the negative side, interpreted programs can only run on the computers that have the respective interpreters.

**Python Data Types:**

3.1.1. Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, \* and / work just like in most other languages (for example, Pascal or C); parentheses (()) can be used for grouping. For example:

>>>

**>>>** 2 + 2

4

**>>>** 50 - 5\*6

20

**>>>** (50 - 5\*6) / 4

5.0

**>>>** 8 / 5 *# division always returns a floating point number*

1.6

The integer numbers (e.g. 2, 4, 20) have type [int](https://docs.python.org/3/library/functions.html#int), the ones with a fractional part (e.g. 5.0, 1.6) have type[float](https://docs.python.org/3/library/functions.html#float). We will see more about numeric types later in the tutorial.

Division (/) always returns a float. To do [floor division](https://docs.python.org/3/glossary.html#term-floor-division) and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use %:

>>>

**>>>** 17 / 3 *# classic division returns a float*

5.666666666666667

>>>

**>>>** 17 // 3 *# floor division discards the fractional part*

5

**>>>** 17 % 3 *# the % operator returns the remainder of the division*

2

**>>>** 5 \* 3 + 2 *# result \* divisor + remainder*

17

With Python, it is possible to use the \*\* operator to calculate powers [1](https://docs.python.org/3/tutorial/introduction.html#id3):

>>>

**>>>** 5 \*\* 2 *# 5 squared*

25

**>>>** 2 \*\* 7 *# 2 to the power of 7*

128

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

>>>

**>>>** width = 20

**>>>** height = 5 \* 9

**>>>** width \* height

900

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

>>>

**>>>** n *# try to access an undefined variable*

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'n' is not defined

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

>>>

**>>>** 4 \* 3.75 - 1

14.0

In interactive mode, the last printed expression is assigned to the variable \_. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

>>>

**>>>** tax = 12.5 / 100

**>>>** price = 100.50

**>>>** price \* tax

12.5625

**>>>** price + \_

113.0625

**>>>** round(\_, 2)

113.06

This variable should be treated as read-only by the user. Don’t explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to [int](https://docs.python.org/3/library/functions.html#int) and [float](https://docs.python.org/3/library/functions.html#float), Python supports other types of numbers, such as [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) and [Fraction](https://docs.python.org/3/library/fractions.html#fractions.Fraction). Python also has built-in support for [complex numbers](https://docs.python.org/3/library/stdtypes.html#typesnumeric), and uses the j or J suffix to indicate the imaginary part (e.g. 3+5j).

3.1.2. Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result [2](https://docs.python.org/3/tutorial/introduction.html#id4). \ can be used to escape quotes:

>>>

**>>>** 'spam eggs' *# single quotes*

'spam eggs'

**>>>** 'doesn**\'**t' *# use \' to escape the single quote...*

"doesn't"

**>>>** "doesn't" *# ...or use double quotes instead*

"doesn't"

**>>>** '"Yes," they said.'

'"Yes," they said.'

**>>>** "**\"**Yes,**\"** they said."

'"Yes," they said.'

**>>>** '"Isn**\'**t," they said.'

'"Isn\'t," they said.'

In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes. The [print()](https://docs.python.org/3/library/functions.html#print) function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

>>>

**>>>** '"Isn**\'**t," they said.'

'"Isn\'t," they said.'

**>>>** print('"Isn**\'**t," they said.')

"Isn't," they said.

**>>>** s = 'First line.**\n**Second line.' *# \n means newline*

**>>>** s *# without print(), \n is included in the output*

'First line.\nSecond line.'

**>>>** print(s) *# with print(), \n produces a new line*

First line.

Second line.

If you don’t want characters prefaced by \ to be interpreted as special characters, you can use *raw strings* by adding an r before the first quote:

>>>

**>>>** print('C:\some**\n**ame') *# here \n means newline!*

C:\some

ame

**>>>** print(r'C:\some\name') *# note the r before the quote*

C:\some\name

String literals can span multiple lines. One way is using triple-quotes: """...""" or '''...'''. End of lines are automatically included in the string, but it’s possible to prevent this by adding a \ at the end of the line. The following example:

print("""**\**

Usage: thingy [OPTIONS]

-h Display this usage message

-H hostname Hostname to connect to

""")

produces the following output (note that the initial newline is not included):

Usage: thingy [OPTIONS]

-h Display this usage message

-H hostname Hostname to connect to

Strings can be concatenated (glued together) with the + operator, and repeated with \*:

>>>

**>>>** *# 3 times 'un', followed by 'ium'*

**>>>** 3 \* 'un' + 'ium'

'unununium'

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

>>>

**>>>** 'Py' 'thon'

'Python'

This feature is particularly useful when you want to break long strings:

>>>

**>>>** text = ('Put several strings within parentheses '

**...**  'to have them joined together.')

**>>>** text

'Put several strings within parentheses to have them joined together.'

This only works with two literals though, not with variables or expressions:

>>>

**>>>** prefix = 'Py'

**>>>** prefix 'thon' *# can't concatenate a variable and a string literal*

File "<stdin>", line 1

prefix 'thon'

^

SyntaxError: invalid syntax

**>>>** ('un' \* 3) 'ium'

File "<stdin>", line 1

('un' \* 3) 'ium'

^

SyntaxError: invalid syntax

If you want to concatenate variables or a variable and a literal, use +:

>>>

**>>>** prefix + 'thon'

'Python'

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

>>>

**>>>** word = 'Python'

**>>>** word[0] *# character in position 0*

'P'

**>>>** word[5] *# character in position 5*

'n'

Indices may also be negative numbers, to start counting from the right:

>>>

**>>>** word[-1] *# last character*

'n'

**>>>** word[-2] *# second-last character*

'o'

**>>>** word[-6]

'P'

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing*allows you to obtain substring:

>>>

**>>>** word[0:2] *# characters from position 0 (included) to 2 (excluded)*

'Py'

**>>>** word[2:5] *# characters from position 2 (included) to 5 (excluded)*

'tho'

Note how the start is always included, and the end always excluded. This makes sure that s[:i] + s[i:] is always equal to s:

>>>

**>>>** word[:2] + word[2:]

'Python'

**>>>** word[:4] + word[4:]

'Python'

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

>>>

**>>>** word[:2] *# character from the beginning to position 2 (excluded)*

'Py'

**>>>** word[4:] *# characters from position 4 (included) to the end*

'on'

**>>>** word[-2:] *# characters from the second-last (included) to the end*

'on'

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

+---+---+---+---+---+---+

| P | y | t | h | o | n |

+---+---+---+---+---+---+

0 1 2 3 4 5 6

-6 -5 -4 -3 -2 -1

The first row of numbers gives the position of the indices 0…6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of word[1:3] is 2.

Attempting to use an index that is too large will result in an error:

>>>

**>>>** word[42] *# the word only has 6 characters*

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: string index out of range

However, out of range slice indexes are handled gracefully when used for slicing:

>>>

**>>>** word[4:42]

'on'

**>>>** word[42:]

''

Python strings cannot be changed — they are [immutable](https://docs.python.org/3/glossary.html#term-immutable). Therefore, assigning to an indexed position in the string results in an error:

>>>

**>>>** word[0] = 'J'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

**>>>** word[2:] = 'py'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

If you need a different string, you should create a new one:

>>>

**>>>** 'J' + word[1:]

'Jython'

**>>>** word[:2] + 'py'

'Pypy'

The built-in function [len()](https://docs.python.org/3/library/functions.html#len) returns the length of a string:

>>>

**>>>** s = 'supercalifragilisticexpialidocious'

**>>>** len(s)

34

**See also**

[**Text Sequence Type — str**](https://docs.python.org/3/library/stdtypes.html#textseq)

Strings are examples of *sequence types*, and support the common operations supported by such types.

[**String Methods**](https://docs.python.org/3/library/stdtypes.html#string-methods)

Strings support a large number of methods for basic transformations and searching.

[**Formatted string literals**](https://docs.python.org/3/reference/lexical_analysis.html#f-strings)

String literals that have embedded expressions.

[**Format String Syntax**](https://docs.python.org/3/library/string.html#formatstrings)

Information about string formatting with [str.format()](https://docs.python.org/3/library/stdtypes.html#str.format).

[**printf-style String Formatting**](https://docs.python.org/3/library/stdtypes.html#old-string-formatting)

The old formatting operations invoked when strings are the left operand of the % operator are described in more detail here.

3.1.3. Lists

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

>>>

**>>>** squares = [1, 4, 9, 16, 25]

**>>>** squares

[1, 4, 9, 16, 25]

Like strings (and all other built-in [sequence](https://docs.python.org/3/glossary.html#term-sequence) types), lists can be indexed and sliced:

>>>

**>>>** squares[0] *# indexing returns the item*

1

**>>>** squares[-1]

25

**>>>** squares[-3:] *# slicing returns a new list*

[9, 16, 25]

All slice operations return a new list containing the requested elements. This means that the following slice returns a [shallow copy](https://docs.python.org/3/library/copy.html#shallow-vs-deep-copy) of the list:

>>>

**>>>** squares[:]

[1, 4, 9, 16, 25]

Lists also support operations like concatenation:

>>>

**>>>** squares + [36, 49, 64, 81, 100]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Unlike strings, which are [immutable](https://docs.python.org/3/glossary.html#term-immutable), lists are a [mutable](https://docs.python.org/3/glossary.html#term-mutable) type, i.e. it is possible to change their content:

>>>

**>>>** cubes = [1, 8, 27, 65, 125] *# something's wrong here*

**>>>** 4 \*\* 3 *# the cube of 4 is 64, not 65!*

64

**>>>** cubes[3] = 64 *# replace the wrong value*

**>>>** cubes

[1, 8, 27, 64, 125]

You can also add new items at the end of the list, by using the append() *method* (we will see more about methods later):

>>>

**>>>** cubes.append(216) *# add the cube of 6*

**>>>** cubes.append(7 \*\* 3) *# and the cube of 7*

**>>>** cubes

[1, 8, 27, 64, 125, 216, 343]

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

>>>

**>>>** letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

**>>>** letters

['a', 'b', 'c', 'd', 'e', 'f', 'g']

**>>>** *# replace some values*

**>>>** letters[2:5] = ['C', 'D', 'E']

**>>>** letters

['a', 'b', 'C', 'D', 'E', 'f', 'g']

**>>>** *# now remove them*

**>>>** letters[2:5] = []

**>>>** letters

['a', 'b', 'f', 'g']

**>>>** *# clear the list by replacing all the elements with an empty list*

**>>>** letters[:] = []

**>>>** letters

[]

The built-in function [len()](https://docs.python.org/3/library/functions.html#len) also applies to lists:

>>>

**>>>** letters = ['a', 'b', 'c', 'd']

**>>>** len(letters)

4

It is possible to nest lists (create lists containing other lists), for example:

>>>

**>>>** a = ['a', 'b', 'c']

**>>>** n = [1, 2, 3]

**>>>** x = [a, n]

**>>>** x

[['a', 'b', 'c'], [1, 2, 3]]

**>>>** x[0]

['a', 'b', 'c']

**>>>** x[0][1]

'b'

3.2. First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the [Fibonacci series](https://en.wikipedia.org/wiki/Fibonacci_number) as follows:

>>>

**>>>** *# Fibonacci series:*

**...** *# the sum of two elements defines the next*

**...** a, b = 0, 1

**>>> while** a < 10:

**...**  print(a)

**...**  a, b = b, a+b

**...**

0

1

1

2

3

5

8

This example introduces several new features.

* The first line contains a *multiple assignment*: the variables a and b simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.
* The [while](https://docs.python.org/3/reference/compound_stmts.html#while) loop executes as long as the condition (here: a < 10) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) and != (not equal to).
* The *body* of the loop is *indented*: indentation is Python’s way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; all decent text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.
* The [print()](https://docs.python.org/3/library/functions.html#print) function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

>>>

**>>>** i = 256\*256

**>>>** print('The value of i is', i)

The value of i is 65536

The keyword argument *end* can be used to avoid the newline after the output, or end the output with a different string:

>>>

**>>>** a, b = 0, 1

**>>> while** a < 1000:

**...**  print(a, end=',')

**...**  a, b = b, a+b

**...**

0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,

# **Control Flow Tools**

4.1. if Statements

Perhaps the most well-known statement type is the [if](https://docs.python.org/3/reference/compound_stmts.html#if) statement. For example:

>>>

**>>>** x = int(input("Please enter an integer: "))

Please enter an integer: 42

**>>> if** x < 0:

**...**  x = 0

**...**  print('Negative changed to zero')

**... elif** x == 0:

**...**  print('Zero')

**... elif** x == 1:

**...**  print('Single')

**... else**:

**...**  print('More')

**...**

More

There can be zero or more [elif](https://docs.python.org/3/reference/compound_stmts.html#elif) parts, and the [else](https://docs.python.org/3/reference/compound_stmts.html#else) part is optional. The keyword ‘elif’ is short for ‘else if’, and is useful to avoid excessive indentation. An if … elif … elif … sequence is a substitute for the switch orcase statements found in other languages.

4.2. for Statements

The [for](https://docs.python.org/3/reference/compound_stmts.html#for) statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

>>>

**>>>** *# Measure some strings:*

**...** words = ['cat', 'window', 'defenestrate']

**>>> for** w **in** words:

**...**  print(w, len(w))

**...**

cat 3

window 6

defenestrate 12

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straight-forward to loop over a copy of the collection or to create a new collection:

*# Strategy: Iterate over a copy*

**for** user, status **in** users.copy().items():

**if** status == 'inactive':

**del** users[user]

*# Strategy: Create a new collection*

active\_users = {}

**for** user, status **in** users.items():

**if** status == 'active':

active\_users[user] = status

4.3. The [range()](https://docs.python.org/3/library/stdtypes.html#range) Function

If you do need to iterate over a sequence of numbers, the built-in function [range()](https://docs.python.org/3/library/stdtypes.html#range) comes in handy. It generates arithmetic progressions:

>>>

**>>> for** i **in** range(5):

**...**  print(i)

**...**

0

1

2

3

4

The given end point is never part of the generated sequence; range(10) generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

range(5, 10)

5, 6, 7, 8, 9

range(0, 10, 3)

0, 3, 6, 9

range(-10, -100, -30)

-10, -40, -70

To iterate over the indices of a sequence, you can combine [range()](https://docs.python.org/3/library/stdtypes.html#range) and [len()](https://docs.python.org/3/library/functions.html#len) as follows:

>>>

**>>>** a = ['Mary', 'had', 'a', 'little', 'lamb']

**>>> for** i **in** range(len(a)):

**...**  print(i, a[i])

**...**

0 Mary

1 had

2 a

3 little

4 lamb

In most such cases, however, it is convenient to use the [enumerate()](https://docs.python.org/3/library/functions.html#enumerate) function, see [Looping Techniques](https://docs.python.org/3/tutorial/datastructures.html#tut-loopidioms).

A strange thing happens if you just print a range:

>>>

**>>>** print(range(10))

range(0, 10)

In many ways the object returned by [range()](https://docs.python.org/3/library/stdtypes.html#range) behaves as if it is a list, but in fact it isn’t. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn’t really make the list, thus saving space.

We say such an object is [iterable](https://docs.python.org/3/glossary.html#term-iterable), that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the [for](https://docs.python.org/3/reference/compound_stmts.html#for) statement is such a construct, while an example of a function that takes an iterable is [sum()](https://docs.python.org/3/library/functions.html#sum):

>>>

**>>>** sum(range(4)) *# 0 + 1 + 2 + 3*

6

Later we will see more functions that return iterables and take iterables as arguments. Lastly, maybe you are curious about how to get a list from a range. Here is the solution:

>>>

**>>>** list(range(4))

[0, 1, 2, 3]

In chapter [Data Structures](https://docs.python.org/3/tutorial/datastructures.html#tut-structures), we will discuss in more detail about [list()](https://docs.python.org/3/library/stdtypes.html#list).

4.4. break and continue Statements, and else Clauses on Loops

The [break](https://docs.python.org/3/reference/simple_stmts.html#break) statement, like in C, breaks out of the innermost enclosing [for](https://docs.python.org/3/reference/compound_stmts.html#for) or [while](https://docs.python.org/3/reference/compound_stmts.html#while) loop.

Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the iterable (with [for](https://docs.python.org/3/reference/compound_stmts.html#for)) or when the condition becomes false (with [while](https://docs.python.org/3/reference/compound_stmts.html#while)), but not when the loop is terminated by a [break](https://docs.python.org/3/reference/simple_stmts.html#break) statement. This is exemplified by the following loop, which searches for prime numbers:

>>>

**>>> for** n **in** range(2, 10):

**...**  **for** x **in** range(2, n):

**...**  **if** n % x == 0:

**...**  print(n, 'equals', x, '\*', n//x)

**...**  **break**

**...**  **else**:

**...**  *# loop fell through without finding a factor*

**...**  print(n, 'is a prime number')

**...**

2 is a prime number

3 is a prime number

4 equals 2 \* 2

5 is a prime number

6 equals 2 \* 3

7 is a prime number

8 equals 2 \* 4

9 equals 3 \* 3

(Yes, this is the correct code. Look closely: the else clause belongs to the [for](https://docs.python.org/3/reference/compound_stmts.html#for) loop, **not** the [if](https://docs.python.org/3/reference/compound_stmts.html#if) statement.)

When used with a loop, the else clause has more in common with the else clause of a [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement than it does with that of [if](https://docs.python.org/3/reference/compound_stmts.html#if) statements: a [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement’s else clause runs when no exception occurs, and a loop’s else clause runs when no break occurs. For more on the try statement and exceptions, see [Handling Exceptions](https://docs.python.org/3/tutorial/errors.html#tut-handling).

The [continue](https://docs.python.org/3/reference/simple_stmts.html#continue) statement, also borrowed from C, continues with the next iteration of the loop:

>>>

**>>> for** num **in** range(2, 10):

**...**  **if** num % 2 == 0:

**...**  print("Found an even number", num)

**...**  **continue**

**...**  print("Found an odd number", num)

Found an even number 2

Found an odd number 3

Found an even number 4

Found an odd number 5

Found an even number 6

Found an odd number 7

Found an even number 8

Found an odd number 9

4.5. pass Statements

The [pass](https://docs.python.org/3/reference/simple_stmts.html#pass) statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

>>>

**>>> while** **True**:

**...**  **pass** *# Busy-wait for keyboard interrupt (Ctrl+C)*

**...**

This is commonly used for creating minimal classes:

>>>

**>>> class** **MyEmptyClass**:

**...**  **pass**

**...**

Another place [pass](https://docs.python.org/3/reference/simple_stmts.html#pass) can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored:

>>>

**>>> def** initlog(\*args):

**...**  **pass** *# Remember to implement this!*

**...**

4.6. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

>>>

**>>> def** fib(n): *# write Fibonacci series up to n*

**...**  *"""Print a Fibonacci series up to n."""*

**...**  a, b = 0, 1

**...**  **while** a < n:

**...**  print(a, end=' ')

**...**  a, b = b, a+b

**...**  print()

**...**

**>>>** *# Now call the function we just defined:*

**...** fib(2000)

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

The keyword [def](https://docs.python.org/3/reference/compound_stmts.html#def) introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function’s documentation string, or *docstring*. (More about docstrings can be found in the section [Documentation Strings](https://docs.python.org/3/tutorial/controlflow.html#tut-docstrings).) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it’s good practice to include docstrings in code that you write, so make a habit of it.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a [global](https://docs.python.org/3/reference/simple_stmts.html#global)statement, or, for variables of enclosing functions, named in a [nonlocal](https://docs.python.org/3/reference/simple_stmts.html#nonlocal) statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object). [1](https://docs.python.org/3/tutorial/controlflow.html#id2) When a function calls another function, a new local symbol table is created for that call.

A function definition associates the function name with the function object in the current symbol table. The interpreter recognizes the object pointed to by that name as a user-defined function. Other names can also point to that same function object and can also be used to access the function:

>>>

**>>>** fib

<function fib at 10042ed0>

**>>>** f = fib

**>>>** f(100)

0 1 1 2 3 5 8 13 21 34 55 89

Coming from other languages, you might object that fib is not a function but a procedure since it doesn’t return a value. In fact, even functions without a [return](https://docs.python.org/3/reference/simple_stmts.html#return) statement do return a value, albeit a rather boring one. This value is called None (it’s a built-in name). Writing the value None is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using [print()](https://docs.python.org/3/library/functions.html#print):

>>>

**>>>** fib(0)

**>>>** print(fib(0))

None

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

>>>

**>>> def** fib2(n): *# return Fibonacci series up to n*

**...**  *"""Return a list containing the Fibonacci series up to n."""*

**...**  result = []

**...**  a, b = 0, 1

**...**  **while** a < n:

**...**  result.append(a) *# see below*

**...**  a, b = b, a+b

**...**  **return** result

**...**

**>>>** f100 = fib2(100) *# call it*

**>>>** f100 *# write the result*

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

This example, as usual, demonstrates some new Python features:

* The [return](https://docs.python.org/3/reference/simple_stmts.html#return) statement returns with a value from a function. return without an expression argument returns None. Falling off the end of a function also returns None.
* The statement result.append(a) calls a *method* of the list object result. A method is a function that ‘belongs’ to an object and is named obj.methodname, where obj is some object (this may be an expression), and methodname is the name of a method that is defined by the object’s type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*, see [Classes](https://docs.python.org/3/tutorial/classes.html#tut-classes)) The method append() shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to result = result + [a], but more efficient.

4.7. More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.7.1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

**def** ask\_ok(prompt, retries=4, reminder='Please try again!'):

**while** **True**:

ok = input(prompt)

**if** ok **in** ('y', 'ye', 'yes'):

**return** **True**

**if** ok **in** ('n', 'no', 'nop', 'nope'):

**return** **False**

retries = retries - 1

**if** retries < 0:

**raise** ValueError('invalid user response')

print(reminder)

This function can be called in several ways:

* giving only the mandatory argument: ask\_ok('Do you really want to quit?')
* giving one of the optional arguments: ask\_ok('OK to overwrite the file?', 2)
* or even giving all arguments: ask\_ok('OK to overwrite the file?', 2, 'Come on, only yes orno!')

This example also introduces the [in](https://docs.python.org/3/reference/expressions.html#in) keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that

i = 5

**def** f(arg=i):

print(arg)

i = 6

f()

will print 5.

**Important warning:** The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

**def** f(a, L=[]):

L.append(a)

**return** L

print(f(1))

print(f(2))

print(f(3))

This will print

[1]

[1, 2]

[1, 2, 3]

If you don’t want the default to be shared between subsequent calls, you can write the function like this instead:

**def** f(a, L=**None**):

**if** L **is** **None**:

L = []

L.append(a)

**return** L

4.7.2. Keyword Arguments

Functions can also be called using [keyword arguments](https://docs.python.org/3/glossary.html#term-keyword-argument) of the form kwarg=value. For instance, the following function:

**def** parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):

print("-- This parrot wouldn't", action, end=' ')

print("if you put", voltage, "volts through it.")

print("-- Lovely plumage, the", type)

print("-- It's", state, "!")

accepts one required argument (voltage) and three optional arguments (state, action, and type). This function can be called in any of the following ways:

parrot(1000) *# 1 positional argument*

parrot(voltage=1000) *# 1 keyword argument*

parrot(voltage=1000000, action='VOOOOOM') *# 2 keyword arguments*

parrot(action='VOOOOOM', voltage=1000000) *# 2 keyword arguments*

parrot('a million', 'bereft of life', 'jump') *# 3 positional arguments*

parrot('a thousand', state='pushing up the daisies') *# 1 positional, 1 keyword*

but all the following calls would be invalid:

parrot() *# required argument missing*

parrot(voltage=5.0, 'dead') *# non-keyword argument after a keyword argument*

parrot(110, voltage=220) *# duplicate value for the same argument*

parrot(actor='John Cleese') *# unknown keyword argument*

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. actor is not a valid argument for the parrotfunction), and their order is not important. This also includes non-optional arguments (e.g. parrot(voltage=1000) is valid too). No argument may receive a value more than once. Here’s an example that fails due to this restriction:

>>>

**>>> def** function(a):

**...**  **pass**

**...**

**>>>** function(0, a=0)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: function() got multiple values for keyword argument 'a'

When a final formal parameter of the form \*\*name is present, it receives a dictionary (see [Mapping Types — dict](https://docs.python.org/3/library/stdtypes.html#typesmapping)) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form \*name (described in the next subsection) which receives a [tuple](https://docs.python.org/3/tutorial/datastructures.html#tut-tuples) containing the positional arguments beyond the formal parameter list. (\*name must occur before \*\*name.) For example, if we define a function like this:

**def** cheeseshop(kind, \*arguments, \*\*keywords):

print("-- Do you have any", kind, "?")

print("-- I'm sorry, we're all out of", kind)

**for** arg **in** arguments:

print(arg)

print("-" \* 40)

**for** kw **in** keywords:

print(kw, ":", keywords[kw])

It could be called like this:

cheeseshop("Limburger", "It's very runny, sir.",

"It's really very, VERY runny, sir.",

shopkeeper="Michael Palin",

client="John Cleese",

sketch="Cheese Shop Sketch")

and of course it would print:

-- Do you have any Limburger ?

-- I'm sorry, we're all out of Limburger

It's very runny, sir.

It's really very, VERY runny, sir.

----------------------------------------

shopkeeper : Michael Palin

client : John Cleese

sketch : Cheese Shop Sketch

Note that the order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.

4.7.3. Special parameters

By default, arguments may be passed to a Python function either by position or explicitly by keyword. For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

A function definition may look like:

def f(pos1, pos2, /, pos\_or\_kwd, \*, kwd1, kwd2):

----------- ---------- ----------

| | |

| Positional or keyword |

| - Keyword only

-- Positional only

where / and \* are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only. Keyword parameters are also referred to as named parameters.

4.7.3.1. Positional-or-Keyword Arguments

If / and \* are not present in the function definition, arguments may be passed to a function by position or by keyword.

4.7.3.2. Positional-Only Parameters

Looking at this in a bit more detail, it is possible to mark certain parameters as *positional-only*. If *positional-only*, the parameters’ order matters, and the parameters cannot be passed by keyword. Positional-only parameters are placed before a / (forward-slash). The / is used to logically separate the positional-only parameters from the rest of the parameters. If there is no / in the function definition, there are no positional-only parameters.

Parameters following the / may be *positional-or-keyword* or *keyword-only*.

4.7.3.3. Keyword-Only Arguments

To mark parameters as *keyword-only*, indicating the parameters must be passed by keyword argument, place an \* in the arguments list just before the first *keyword-only* parameter.

4.7.3.4. Function Examples

Consider the following example function definitions paying close attention to the markers / and \*:

>>>

**>>> def** standard\_arg(arg):

**...**  print(arg)

**...**

**>>> def** pos\_only\_arg(arg, /):

**...**  print(arg)

**...**

**>>> def** kwd\_only\_arg(\*, arg):

**...**  print(arg)

**...**

**>>> def** combined\_example(pos\_only, /, standard, \*, kwd\_only):

**...**  print(pos\_only, standard, kwd\_only)

The first function definition, standard\_arg, the most familiar form, places no restrictions on the calling convention and arguments may be passed by position or keyword:

>>>

**>>>** standard\_arg(2)

2

**>>>** standard\_arg(arg=2)

2

The second function pos\_only\_arg is restricted to only use positional parameters as there is a / in the function definition:

>>>

**>>>** pos\_only\_arg(1)

1

**>>>** pos\_only\_arg(arg=1)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: pos\_only\_arg() got an unexpected keyword argument 'arg'

The third function kwd\_only\_args only allows keyword arguments as indicated by a \* in the function definition:

>>>

**>>>** kwd\_only\_arg(3)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: kwd\_only\_arg() takes 0 positional arguments but 1 was given

**>>>** kwd\_only\_arg(arg=3)

3

And the last uses all three calling conventions in the same function definition:

>>>

**>>>** combined\_example(1, 2, 3)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: combined\_example() takes 2 positional arguments but 3 were given

**>>>** combined\_example(1, 2, kwd\_only=3)

1 2 3

**>>>** combined\_example(1, standard=2, kwd\_only=3)

1 2 3

**>>>** combined\_example(pos\_only=1, standard=2, kwd\_only=3)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: combined\_example() got an unexpected keyword argument 'pos\_only'

Finally, consider this function definition which has a potential collision between the positional argument name and \*\*kwds which has name as a key:

**def** foo(name, \*\*kwds):

**return** 'name' **in** kwds

There is no possible call that will make it return True as the keyword 'name' will always bind to the first parameter. For example:

>>>

**>>>** foo(1, \*\*{'name': 2})

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: foo() got multiple values for argument 'name'

>>>

But using / (positional only arguments), it is possible since it allows name as a positional argument and 'name'as a key in the keyword arguments:

**def** foo(name, /, \*\*kwds):

**return** 'name' **in** kwds

>>> foo(1, \*\*{'name': 2})

**True**

In other words, the names of positional-only parameters can be used in \*\*kwds without ambiguity.

4.7.3.5. Recap

The use case will determine which parameters to use in the function definition:

**def** f(pos1, pos2, /, pos\_or\_kwd, \*, kwd1, kwd2):

As guidance:

* Use positional-only if you want the name of the parameters to not be available to the user. This is useful when parameter names have no real meaning, if you want to enforce the order of the arguments when the function is called or if you need to take some positional parameters and arbitrary keywords.
* Use keyword-only when names have meaning and the function definition is more understandable by being explicit with names or you want to prevent users relying on the position of the argument being passed.
* For an API, use positional-only to prevent breaking API changes if the parameter’s name is modified in the future.

4.7.4. Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see [Tuples and Sequences](https://docs.python.org/3/tutorial/datastructures.html#tut-tuples)). Before the variable number of arguments, zero or more normal arguments may occur.

**def** write\_multiple\_items(file, separator, \*args):

file.write(separator.join(args))

Normally, these variadic arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the \*argsparameter are ‘keyword-only’ arguments, meaning that they can only be used as keywords rather than positional arguments.

>>>

**>>> def** concat(\*args, sep="/"):

**...**  **return** sep.join(args)

**...**

**>>>** concat("earth", "mars", "venus")

'earth/mars/venus'

**>>>** concat("earth", "mars", "venus", sep=".")

'earth.mars.venus'

4.7.5. Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in [range()](https://docs.python.org/3/library/stdtypes.html#range) function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the \*-operator to unpack the arguments out of a list or tuple:

>>>

**>>>** list(range(3, 6)) *# normal call with separate arguments*

[3, 4, 5]

**>>>** args = [3, 6]

**>>>** list(range(\*args)) *# call with arguments unpacked from a list*

[3, 4, 5]

In the same fashion, dictionaries can deliver keyword arguments with the \*\*-operator:

>>>

**>>> def** parrot(voltage, state='a stiff', action='voom'):

**...**  print("-- This parrot wouldn't", action, end=' ')

**...**  print("if you put", voltage, "volts through it.", end=' ')

**...**  print("E's", state, "!")

**...**

**>>>** d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}

**>>>** parrot(\*\*d)

-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !

4.7.6. Lambda Expressions

Small anonymous functions can be created with the [lambda](https://docs.python.org/3/reference/expressions.html#lambda) keyword. This function returns the sum of its two arguments: lambda a, b: a+b. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

>>>

**>>> def** make\_incrementor(n):

**...**  **return** **lambda** x: x + n

**...**

**>>>** f = make\_incrementor(42)

**>>>** f(0)

42

**>>>** f(1)

43

The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument:

>>>

**>>>** pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]

**>>>** pairs.sort(key=**lambda** pair: pair[1])

**>>>** pairs

[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]

4.7.7. Documentation Strings

Here are some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object’s purpose. For brevity, it should not explicitly state the object’s name or type, since these are available by other means (except if the name happens to be a verb describing a function’s operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object’s calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line *after* the first line of the string determines the amount of indentation for the entire documentation string. (We can’t use the first line since it is generally adjacent to the string’s opening quotes so its indentation is not apparent in the string literal.) Whitespace “equivalent” to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

>>>

**>>> def** my\_function():

**...**  *"""Do nothing, but document it.*

**...**

**...**  *No, really, it doesn't do anything.*

**...**  *"""*

**...**  **pass**

**...**

**>>>** print(my\_function.\_\_doc\_\_)

Do nothing, but document it.

No, really, it doesn't do anything.

4.7.8. Function Annotations

[Function annotations](https://docs.python.org/3/reference/compound_stmts.html#function) are completely optional metadata information about the types used by user-defined functions (see [**PEP 3107**](https://www.python.org/dev/peps/pep-3107) and [**PEP 484**](https://www.python.org/dev/peps/pep-0484) for more information).

[Annotations](https://docs.python.org/3/glossary.html#term-function-annotation) are stored in the \_\_annotations\_\_ attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal ->, followed by an expression, between the parameter list and the colon denoting the end of the [def](https://docs.python.org/3/reference/compound_stmts.html#def) statement. The following example has a positional argument, a keyword argument, and the return value annotated:

>>>

**>>> def** f(ham: str, eggs: str = 'eggs') -> str:

**...**  print("Annotations:", f.\_\_annotations\_\_)

**...**  print("Arguments:", ham, eggs)

**...**  **return** ham + ' and ' + eggs

**...**

**>>>** f('spam')

Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}

Arguments: spam eggs

'spam and eggs'

4.8. Intermezzo: Coding Style

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about *coding style*. Most languages can be written (or more concise, *formatted*) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

For Python, [**PEP 8**](https://www.python.org/dev/peps/pep-0008) has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

* Use 4-space indentation, and no tabs.

4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.

* Wrap lines so that they don’t exceed 79 characters.

This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.

* Use blank lines to separate functions and classes, and larger blocks of code inside functions.
* When possible, put comments on a line of their own.
* Use docstrings.
* Use spaces around operators and after commas, but not directly inside bracketing constructs: a = f(1,2) + g(3, 4).
* Name your classes and functions consistently; the convention is to use UpperCamelCase for classes and lowercase\_with\_underscores for functions and methods. Always use self as the name for the first method argument (see [A First Look at Classes](https://docs.python.org/3/tutorial/classes.html#tut-firstclasses) for more on classes and methods).
* Don’t use fancy encodings if your code is meant to be used in international environments. Python’s default, UTF-8, or even plain ASCII work best in any case.
* Likewise, don’t use non-ASCII characters in identifiers if there is only the slightest chance people speaking a different language will read or maintain the code.