Pierre Nugues

Language Processing with Python

Draft document: Do not redistribute

November 4, 2019

Springer

II

November 4, 2019

1

A Tour of Python

1.1 Why Python?

Python has become the most popular scripting language. Perl, Ruby, or Lua have similar qualities and goals, sport active developer communities, and have application niches. Nonetheless, none of them can claim the spread and universality of Python. Python's rigorous design, ascetic syntax, simplicity, and the availability of scores of libraries made it the language chosen by almost 70% of the American universities for teaching computer science (Guo, 2014). This makes Python unescapable when it comes to natural language processing.

We used Perl in the first editions of this book as it featured rich regular expressions and a support for Unicode; they are still unsurpassed. Python later adopted these features to a point that now makes the lead of Perl in these areas less significant. And the programming style conveyed by Python, both Spartan and elegant, eventually prevailed. The purpose of this chapter is to provide a quick introduction to Python's syntax to readers with some knowledge in programming.

Python comes in two flavors: Python 2 and Python 3. In this book, we only use Python 3 as Python 2 does not properly support Unicode. Moreover, given a problem, there are often many ways to solve it. Among the possible constructs, some are more conformant to the spirit of Python. van Rossum et al. (2013) wrote a guide on the Pythonic coding style that we try to follow in this book.

1.2 The Read, Evaluate, and Print Loop

Once installed, we start Python either from a terminal or an integrated development environment (IDE). Python uses a loop that reads the user's statements, evaluates them, and prints the results (REPL). Python uses a prompt, the >>> sequence, to tell it is ready to accept a command. Here is an example, where we create variables and assign them with values, numbers and strings, and carry out a few arithmetic operations:

```
$ python
>>> a = 1                 We create variable a and assign it with 1
>>> b = 2                 We create b and assign it with 2
>>> b + 1                 We add 1 to b
3                         And Python returns the result
>>> c = a / (b + 1)       We carry out a computation and assign it to c
>>> c                     We print c
0.3333333333333333        We create text and assign it with a string
>>> text = 'Result:'      And we print both text and c
>>> print(text, c)
Result: 0.3333333333333333
>>> quit()
$
```

We can also write these statements in a file, `first.py`:

```
# A first program
a = 1
b = 2
c = a / (b + 1)
text = 'Result:'
print(text, c)
```

and execute it by typing:

```
$ python first.py
Result: 0.3333333333333333
```

1.3 Introductory Programs

Like all the structured languages, programs in Python consist of blocks, i.e. bodies of contiguous statements together with control statements. In Python, these blocks are defined by an identical indentation: We create a new block by adding an indentation of four spaces from the previous line. This indentation is decreased by the same number of spaces to mean the end of the block.

The program below uses a loop to print the numbers of a list. The loop starts with the `for` and `in` statements ended with a colon. After this statement, we add an indentation of four spaces to define the body of the loop: The statements executed by this loop. We remove the indentation when the block has ended:

```
for i in [1, 2, 3, 4, 5, 6]:
    print(i)
print('Done')
```

Table 1.1. Summary of the main Python operators

Unary operators	`not`	Logical not
	`~`	Binary not
	+ and -	Arithmetic plus sign and negation
Arithmetic operators	*, /, **	Multiplication, division, and exponentiation
	// and %	Floor division and modulo
	+ and -	Addition and subtraction
String operator	+	String concatenation
Comparison operators	> and <	Greater than and less than
	>= and <=	Greater than or equal and less than or equal
	== and !=	Equal and not equal
Logical operators	`and`	Logical and
	`or`	Logical or
Shift operators	<< and >>	Shift left and shift right
Binary bitwise operators	&, |, ^	and, or, xor

The next program introduces a condition with the `if` and `else` statements, also ended with a colon, and the modulo operator, `%`, to print the odd and even numbers:

```
for i in [1, 2, 3, 4, 5, 6]:
    if i % 2 == 0:
        print('Even:', i)
    else:
        print('Odd:', i)
print('Done')
```

Table 1.1 shows common operators in Python.

1.4 Strings

A string in Python is a sequence of characters or symbols enclosed within matching single, double, or triple quotes as, respectively, `'my string'`, `"my string"`, and `"""my string"""`. We use triple quotes to create strings spanning multiple lines as with:

```
iliad = """Sing, O goddess, the anger of Achilles son of
Peleus, that brought countless ills upon the Achaeans."""
```

where the string is stored in the `iliad` variable.

In the example above, the string includes a new line delimiter, `'\n'`, between *of* and *Peleus* to break the line. If, instead, we want to keep the white spaces and just wrap the line so that it fits our text editor, we will use the backslash continuation character, \, as in:

```
iliad2 = 'Sing, O goddess, the anger of Achilles son of \
Peleus, that brought countless ills upon the Achaeans.'
```

where the line break is ignored and iliad2 is equivalent to one single line. We can use any type of quote then.

1.4.1 String Index

We access the characters in a string using their index enclosed in square brackets, starting at 0:

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
alphabet[0]          # 'a'
alphabet[1]          # 'b'
alphabet[25]         # 'z'
```

We can use negative indices, that start from the end of the string:

```
alphabet[-1]         # the last character of a string: 'z'
alphabet[-2]         # the second last: 'y'
alphabet[-26]        # 'a'
```

An index outside the range of the string, like `alphabet[27]`, will throw an index error.

The length of a string is given by the `len()` function:

```
len(alphabet)        # 26
```

There is no limit to this length; we can use them to store a whole corpus, provided that our machine has enough memory.

Once created, strings are immutable and we cannot change their content:

```
alphabet[0] = 'b'    # throws an error
```

1.4.2 String Operations and Functions

Strings come with a set of built-in operators and functions. We concatenate and repeat strings using + and * as in:

```
'abc' + 'def'    # 'abcdef'
'abc' * 3        # 'abcabcabc'
```

The `join()` function is an alternative to +. It is called by a string with a list as argument: `str.join(list)`. It concatenates the elements of the list with the calling string, possibly empty, placed in-between:

```
''.join(['abc', 'def', 'ghi'])   # equivalent to a +:
                                 # 'abcdefghi'
' '.join(['abc', 'def', 'ghi'])  # places a space between the
                                 # elements: 'abc def ghi'
', '.join(['abc', 'def', 'ghi']) # 'abc, def, ghi'
```

We set a string in uppercase letters with `str.upper()` and in lowercase with `str.lower()`:

```
accented_e = 'eéèêë'
accented_e.upper()     # 'EÉÈÊË'
accented_E = 'EÉÈÊË'
accented_E.lower()     # 'eéèêë'
```

We search and replace substrings in strings using `str.find()` and `str.replace()`. `str.find()` returns the index of the first occurrence of the substring or -1, if not found, while `replace()` replaces all the occurrences of the substring and returns a new string:

```
alphabet.find('def')        # 3
alphabet.find('é')          # -1
alphabet.replace('abc', 'αβγ') # 'αβγdefghijklmnopqrstuvwxyz'
```

We can iterate over the characters of a string using a `for in` loop, and for instance extract all its vowels as in:

```
text_vowels = ''
for i in iliad:
    if i in 'aeiou':
        text_vowels = text_vowels + i
print(text_vowels)   # 'ioeeaeoieooeeuaououeiuoeaea'
```

We can abridge the statement:

```
text_vowels = text_vowels + i
```

into

```
text_vowels +=  i
```

as well as for all the arithmetic operators: `-=`, `*=`, `/=`, `**=`, and `%=`.

1.4.3 Slices

We can extract substrings of a string using **slices**: A range defined by a start and an end index, `[start:end]`, where the slice will include all the characters from index `start` up to index `end - 1`:

```
alphabet[0:3]    # the three first letters of alphabet: 'abc'
alphabet[:3]     # equivalent to alphabet[0:3]
alphabet[3:6]    # substring from index 3 to index 5: 'def'
alphabet[-3:]    # the three last letters of alphabet: 'xyz'
alphabet[10:-10] # 'klmnop'
alphabet[:]      # all the letters: 'a...z'
```

As the end index is excluded from the slice,

```
alphabet[:i] + alphabet[i:]
```

Table 1.2. Escape sequences in Python

Sequence	Description	Sequence	Description
`\t`	Tabulation	`\100`	Octal ASCII, three digits, here @
`\n`	New line	`\x40`	Hexadecimal ASCII, two digits, here @
`\r`	Carriage return	`\N{COMMERCIAL AT}`	Unicode name, here @
`\f`	Form feed	`\u0152`	Unicode code point, 16 bits, here Œ
`\b`	Backspace	`\U00000152`	Unicode code point, 32 bits, here Œ
`\a`	Bell		
`\'`	Single quote		
`\"`	Double quote		
`\\`	Backslash		

is always equal to the original string, whatever the value of `i`.

In addition to the start and the end, we can add a step using the syntax `[start:end:step]`. With a step of 2, we extract every second letter:

```
alphabet[0::2]   # acegikmoqzuwy
```

1.4.4 Special Characters

The characters in the strings are interpreted literally by Python, except the quotes and backslashes. To create strings containing these two characters, Python defines two *escape sequences*: `\'` to represent a quote and `\\` to represent a backslash as in:

```
'Python\'s strings'    # "Python's strings"
```

This expression creates the string *Python's strings*; the backslash escape character tells Python to read the quote literally instead of interpreting it as an end-of-string delimiter.

We can also use literal single quotes inside a string delimited by double quotes as in:

```
"Python's strings"     # "Python's strings"
```

Python interpolates certain backslashed sequences, like `\n` or `\t`. For example, `\n` is interpreted as a new line and `\t` as a tabulation. Table 1.2 shows a list of escape sequences with their meaning.

The right column in Table 1.2 lists the numerical representations of characters using the ASCII and Unicode standards. The `\N{name}` name and `\uxxxx` and `\Uxxxxxxxx` sequences enable us to designate any character, like *Ö* and *Œ*, by its Unicode name, respectively, `\N{LATIN CAPITAL LETTER O WITH DIAERESIS}` and `\N{LATIN CAPITAL LIGATURE OE}`, or its code point, `\u00D6` and `\u0152`. We review both the ASCII and Unicode schemes in Chap. **??**. We can also use `\ooo` octal and `\xhh` hexadecimal sequences:

```
'\N{COMMERCIAL AT}'     # '@'
'\x40'                  # '@'
'\100'                  # '@'
'\u0152'                # 'Œ'
```

If we want to treat backslashes as normal characters, we add the `r` prefix (raw) to the string as in:

```
r'\N{COMMERCIAL AT}'     # '\\N{COMMERCIAL AT}'
r'\x40'                  # '\\x40'
r'\100'                  # '\\100'
r'\u0152'                # '\\u0152'
```

These raw strings will be useful to write regular expressions; see Sect. **??**.

1.4.5 Formatting Strings

Python can interpolate variables inside strings. This process is called formatting and uses the `str.format()` function. The positions of the variables in the string are given by curly braces: `{}` that will be replaced by the arguments in `format()` in the same order as in:

```
begin = 'my'
'{} string {}'.format(begin, 'is empty')
                                    # 'my string is empty'
```

`format()` has many options like reordering the arguments through indices:

```
begin = 'my'
'{1} string {0}'.format('is empty', begin)
                                    # 'my string is empty'
```

If the input string contains braces, we escape them by doubling them: `{{` for a literal `{` and `}}` for `}`.

1.5 Data Types

Python has a rich set of data types. The primitive types include:

- The Boolean type, `bool`, with the values `True` and `False`;
- The None type with the `None` value as unique member, equivalent to null in C or Java;
- The integers, `int`;
- The floating point numbers, `float`.

We have also seen the `str` string data type consisting of sequences of Unicode characters.

We return the type of a value with the `type()` function:

```
type(alphabet)      # <class 'str'>
type(12)            # <class 'int'>
type('12')          # <class 'str'>
type(12.0)          # <class 'float'>
type(True)          # <class 'bool'>
type(1 < 2)         # <class 'bool'>
type(None)          # <class 'NoneType'>
```

Python supports the conversion of types using a function with the type name as `int()` or `str()`. When the conversion is not possible, Python throws an error:

```
int('12')           # 12
str(12)             # '12'
int('12.0')         # ValueError
int(alphabet)       # ValueError
int(True)           # 1
int(False)          # 0
bool(7)             # True
bool(0)             # False
bool(None)          # False
```

Like in other programming languages, the Boolean `True` and `False` values have synonyms in the other types:

False: `int`: 0, `float`: 0.0, in the none type, `None`. The empty data structures in general are synonyms of False as the empty string (`str`) '' and the empty list, `[]`;
True: The rest.

1.6 Data Structures

1.6.1 Lists

Lists in Python are data structures that can hold any number of elements of any type. Like in strings, each element has a position, where we can read data using the position index. We can also write data to a specific index and a list grows or shrinks automatically when elements are appended, inserted, or deleted. Python manages the memory without any intervention from the programmer.

Here are some examples of lists:

```
list1 = []          # An empty list
list1 = list()      # Another way to create an empty list
list2 = [1, 2, 3]   # List containing 1, 2, and 3
```

Reading or writing a value to a position of the list is done using its index between square brackets starting from 0. If an element is read or assigned to a position that does not exist, Python returns an index error:

```
list2[1]          # 2
list2[1] = 8
list2             # [1, 8, 3]
list2[4]          # Index error
```

Lists can contain elements of different types:

```
var1 = 3.14
var2 = 'my string'
list3 = [1, var1, 'Prolog', var2]
list3             # [1, 3.14, 'Prolog', 'my string']
```

As with strings, we can extract sublists from a list using slices. The syntax is the same, but unlike strings, we can also assign a list to a slice:

```
list3[1:3]       # [3.14, 'Prolog']
list3[1:3] = [2.72, 'Perl', 'Python']
list3            # [1, 2.72, 'Perl', 'Python', 'my string']
```

We can create lists of lists:

```
list4 = [list2, list3]
    # [[1, 8, 3], [1, 2.72, 'Perl', 'Python', 'my string']]
```

where we access the elements of the inner lists with a sequence of indices between square brackets:

```
list4[0][1]     # 8
list4[1][3]     # 'Python'
```

We can also assign complete list to a variable and a list to a list of variables as in:

```
list5 = list2
[v1, v2, v3] = list5
```

where `list5` contains a copy of `list2`, and `v1`, `v2`, `v3` contain, respectively, 1, 8, and 3.

1.6.2 Built-in List Operations and Functions

Lists have built-in operators and functions. Like for strings, we can use the `+` and `*` operators to concatenate and repeat lists:

```
list2                     # [1, 8, 3]
list3[:-1]                # [1, 2.72, 'Perl', 'Python']
[1, 2, 3] + ['a', 'b']    # [1, 2, 3, 'a', 'b']
list2[:2] + list3[2:-1]   # [1, 8, 'Perl', 'Python']
list2 * 2                 # [1, 8, 3, 1, 8, 3]
[0.0] * 4                 # Initializes a list of four 0.0s
                          # [0.0, 0.0, 0.0, 0.0]
```

In addition to operators, lists have functions that include:

- `list.extend(elements)` that extends the list with the elements of `elements` passed as argument;
- `list.append(element)` that appends `element` to the end of the list;
- `list.insert(idx, element)` that inserts `element` at index `idx`;
- `list.remove(value)` that removes the first occurrence of value;
- `list.pop(i)`, that removes the element at index `i` and returns its value; If there is no index, `list.pop()` takes the last element in the list;
- `del list[i]`, a statement that also removes the element at index `i`. In addition, `del` can remove slices, clear the whole list, or delete the `list` variable;
- `len()`, a function that returns the length of `list`;
- `list.sort()` that sorts the list;
- `sorted()` a function that returns a sorted list.

A few examples:

```
list2                    # [1, 8, 3]
list2[1] = 2             # [1, 2, 3]
len(list2)               # 3
list2.extend([4, 5])     # [1, 2, 3, 4, 5]
list2.append(6)          # [1, 2, 3, 4, 5, 6]
list2.append([7, 8])     # [1, 2, 3, 4, 5, 6, [7, 8]]
list2.pop(-1)            # [1, 2, 3, 4, 5, 6]
list2.remove(1)          # [2, 3, 4, 5, 6]
list2.insert(0, 'a')     # ['a', 2, 3, 4, 5, 6]
```

To know all the functions associated with a type, we can use `dir()`, as in:

```
dir(list)
```

or

```
dir(str)
```

To have help on a specific type or function, we can use `help` as in:

```
help(list)
```

and

```
help(list.append)
```

or read the online documentation.

1.6.3 Tuples

Tuples are sequences enclosed in parentheses. They are very similar to lists, except that they are immutable. Once created, we access the elements of a tuple, including slices, using the same notation as with the lists.

```
tuple1 = ()          # An empty tuple
tuple1 = tuple()     # Another way to create an empty tuple
tuple2 = (1, 2, 3, 4)
tuple2[3]            # 4
tuple2[1:4]          # (2, 3, 4)
tuple2[3] = 8        # Type error: Tuples are immutable
```

Parentheses enclosing one item could be ambiguous as (1), for example, as it already denotes an arithmetic expression. That is why tuples of one item require a trailing comma:

```
type((1))    # <class 'int'>
             # Arithmetic expression corresponding to integer 1
type((1,))   # <class 'tuple'>
             # A tuple consisting of one item: integer 1
```

We can convert lists to tuples and tuples to lists:

```
list6 = ['a', 'b', 'c']
tuple3 = tuple(list6) # conversion to a tuple: ('a', 'b', 'c')
type(tuple3)          # <class 'tuple'>
list7 = list(tuple2)  # [1, 2, 3, 4]
tuple([1])            # (1,)
                      # conversion to a tuple of one item
list((1,))            # [1]
                      # conversion to a list of one item
```

Tuple can include elements of different types. If an inner element is mutable, we can change its value as in:

```
tuple4 = (tuple2, list6) # ((1, 2, 3, 4), ['a', 'b', 'c'])
tuple4[0]                # (1, 2, 3, 4),
tuple4[1]                # ['a', 'b', 'c']
tuple4[0][2]             # 3
tuple4[1][1]             # 'b'
tuple4[1][1] = 'β'       # ((1, 2, 3, 4), ['a', 'β', 'c'])
```

1.6.4 Sets

Sets are collections that have no duplicates. We create a set with a sequence enclosed in curly braces or an empty set with the set() function. We can then add and remove elements with the add() and remove() functions:

```
set1 = set()                        # An empty set
set2 = {'a', 'b', 'c', 'c', 'b'}    # {'a', 'b', 'c'}
set2.add('d')                       # {'a', 'b', 'c', 'd'}
set2.remove('a')                    # {'b', 'c', 'd'}
```

Sets are useful to extract the unique elements of lists or strings as in:

```
list8 = ['a', 'b', 'c', 'c', 'b']
set3 = set(list8)                  # {'a', 'b', 'c'}
iliad_chars = set(iliad.lower())
        # The set of unique characters of the iliad string
```

Sets are unordered. We can create a sorted list of them using `sorted()` as in:

```
>>> sorted(iliad_chars)
['\n', ' ', '"', ',', '.', 'a', 'b', 'c', 'd', 'e', 'f',
  'g', 'h', 'i', 'l', 'n', 'o', 'p', 'r', 's', 't', 'u']
```

1.6.5 Built-in Set Functions

The set library includes the classical set operations:

- `set1.intersection(set2, ...)`
- `set1.union(set2, ...)`
- `set1.difference(set2, ...)`
- `set1.symmetric_difference(set2)`
- `set1.issuperset(set2)`
- `set1.issubset(set2)`

A few examples:

```
set2.intersection(set3)            # {'c', 'b'}
set2.union(set3)                   # {'d', 'b', 'a', 'c'}
set2.symmetric_difference(set3)    # {'a', 'd'}
set2.issubset(set3)                # False
iliad_chars.intersection(set(alphabet))
      # characters of the iliad string that are letters:
      # {'a', 's', 'g', 'p', 'u', 'h', 'c', 'l', 'i',
      #  'd', 'o', 'e', 'b', 't', 'f', 'r', 'n'}
```

1.6.6 Dictionaries

Dictionaries are collections, where the values are indexed by keys instead of ordered positions, like in lists or tuples. Counting the words of a text is a very frequent operation in natural language processing, as we will see in the rest of this book. Dictionaries are the most appropriate data structures to carry this out, where we use the keys to store the words and the values to store the counts.

We create a dictionary by assigning it a set of initial key-value pairs, possibly empty, where keys and values are separated by a colon, and then adding keys and values using the same syntax as with the lists. The statements:

```
wordcount = {}          # We create an empty dictionary
wordcount = dict()      # Another way to create a dictionary
wordcount['a'] = 21     # The key 'a' has value 21
wordcount['And'] = 10   # 'And' has value 10
wordcount['the'] = 18
```

create the dictionary wordcount and add three keys: a, And, the, whose values are 21, 10, and 18. We refer to the whole dictionary using the notation wordcount.

```
>>> wordcount
{'the': 18, 'a': 21, 'And': 10}
```

The order of the keys is not defined at run-time and we cannot rely on it.

The values of the resulting dictionary can be accessed by their keys with the same syntax as with lists:

```
wordcount['a']        # 21
wordcount['And']      # 10
```

A dictionary entry is created when a value is assigned to it. Its existence can be tested using the in Boolean function:

```
'And' in wordcount    # True
'is' in wordcount     # False
```

Just like indices for lists, the key must exist to access it, otherwise it generates an error:

```
wordcount['is']       # Key error
```

To access a key in a dictionary without risking an error, we can use the get() function that has a default value if the key is undefined:

- get('And') returns the value of the key or None if undefined;
- get('is', val) returns the value of the key or val if undefined.

as in:

```
wordcount.get('And')   # 10
wordcount.get('is', 0) # 0
wordcount.get('is')    # None
```

Keys can be strings, numbers, or immutable structures. Mutable keys, like a list, will generate an error:

```
my_dict = {}
my_dict[('And', 'the')] = 3  # OK, we use a tuple
my_dict[['And', 'the']] = 3  # Type error:
                             # unhashable type: 'list'
```

1.6.7 Built-in Dictionary Functions

Dictionaries have a set of built-in functions. The most useful ones are:

- keys() returns the keys of a dictionary;
- values() returns the values of a dictionary;
- items() returns the key-value pairs of a dictionary.

A few examples:

```
wordcount.keys()       # dict_keys(['the', 'a', 'And'])
wordcount.values()     # dict_values([18, 21, 10])
wordcount.items()      # dict_items([('the', 18), ('a', 21),
                       # ('And', 10)])
```

1.6.8 Counting the Letters of a Text

Let us finish with a program that counts the letters of a text. We use the `for in` statement to scan the `iliad` text set in lowercase letters; we increment the frequency of the current letter if it is in the dictionary or we set it to 1, if we have not seen it before.

The complete program is:

```
letter_count = {}
for letter in iliad.lower():
    if letter in alphabet:
        if letter in letter_count:
            letter_count[letter] += 1
        else:
            letter_count[letter] = 1
```

resulting in:

```
>>> letter_count
{'g': 4, 's': 10, 'o': 8, 'u': 4, 'h': 6, 'c': 3, 'l': 6,
'a': 6, 't': 6, 'd': 2, 'e': 9, 'b': 1, 'p': 2, 'f': 2,
'r': 2, 'n': 6, 'i': 3}
```

To print the result in alphabetical order, we extract the keys; we sort them; and we print the key-value pairs. We do all this with this loop:

```
for letter in sorted(letter_count.keys()):
    print(letter, letter_count[letter])
```

Running it results in:

```
a 6
b 1
c 3
d 2
e 9
...
```

By default, `sorted()` sorts the elements alphabetically. If we want to sort the letters by frequency, we can use the `key` argument of `sorted()`. `key` specifies a function whose result is used to compare the elements. In our case, we want to compare the frequencies, that is the values of the dictionary. We saw that we extract these values with the `get` method, here `letter_count.get`, and we hence assign it to `key`.

Using `get`, the letters will be sorted from the least frequent to the most frequent. If we want to reverse this order, we use the third argument, `reverse`, a Boolean value, that we set to `True`.

Finally, our new loop:

```
for letter in sorted(letter_count.keys(),
                     key=letter_count.get, reverse=True):
    print(letter, letter_count[letter])
```

produces this output:

```
s 10
e 9
o 8
t 6
h 6
...
```

1.7 Control Structures

In Python, the control flow statements include conditionals, loops, exceptions, and functions. These statements consist of two parts, the header and the suite. The header starts with a keyword like `if`, `for`, or `while` and ends with a colon. The suite consists of the statement sequence controlled by the header; we have seen that the statement in the suite must be indented with four characters.

At this point, we may wonder how we can break expressions in multiple lines, for instance to improve the readability of a long list or long arithmetic operations. The answer is to make use of parentheses, square or curly brackets. A statement inside parentheses or brackets is equivalent to a unique logical line, even if it contains line breaks.

1.7.1 Conditionals

Python expresses conditions with the `if`, `elif`, and `else` statements as in:

```
digits = ’0123456789’
punctuation = ’.,;:?!’

char = ’.’
if char in alphabet:
    print(’Letter’)
elif char in digits:
    print(’Number’)
elif char in punctuation:
```

```
    print('Punctuation')
else:
    print('Other')
```

that will print `Punctuation`.

1.7.2 The `for` Loop

A `for in` loop in Python iterates over the elements of a sequence such as a string or a list. This differs from languages like Perl, C or Java, where the typical `for` iteration is over numbers. If we need to create such loops, Python has the `range(start, stop, step)` function that returns a sequence of numbers. Only one argument is required: `stop`. The variables `start` and `step` will default to 0 and 1.

The next program generates the integers from 0 to 99 and computes their sum:

```
sum = 0
for i in range(100):
    sum += i
print(sum)    # Sum of integers from 0 to 99: 4950
              # Using the built-in sum() function,
              # sum(range(100)) would produce the same result.
```

The `range()` behavior is comparable to that of a list, but as a list will grow with its length, `range()` will use a constant memory. Nonetheless, we can convert a range into a list:

```
list10 = list(range(5))       # [0, 1, 2, 3, 4]
```

We have seen how to iterate over a list and over indices using `range()`. Should we want to iterate over both, we can use the `enumerate()` function. `enumerate()` takes a sequence as argument and returns a sequence of `(index, element)` pairs, where `element` is an element of the sequence and `index`, its index.

We can use `enumerate()` to get the letters of the alphabet and their index with the program:

```
for idx, letter in enumerate(alphabet):
    print(idx, letter)
```

that prints:

```
0 a
1 b
2 c
3 d
4 e
5 f
...
```

Note the parallel assignment of `idx` and `letter`.

1.7.3 The `while` Loop

The `while` loop is an alternative to `for`, although less frequent in Python programs. This loop executes a block of statements as long as a condition is true. We can reformulate the counting `for` loop in Sect 1.7.2 using `while`:

```
sum, i = 0, 0
while i < 100:
    sum += i
    i += 1
print(sum)
```

Another possible structure is to use an infinite loop and a `break` statement to exit the loop:

```
sum, i = 0, 0
while True:
    sum += i
    i += 1
    if i >= 100:
        break
print(sum)
```

Note that it is not possible to assign a variable in the condition of a `while` statement.

1.7.4 Exceptions

Python has a mechanism to handle errors so that they do not stop a program. It uses the `try` and `except` keywords. We saw in Sect. 1.5 that the conversion of the `alphabet` and `'12.0'` strings into integers prints an error and exits the program. We can handle it safely with the `try/except` construct:

```
try:
    int(alphabet)
    int('12.0')
except:
    pass
print('Cleared the exception!')
```

where `pass` is an empty statement serving as a placeholder for the `except` block.

It is also possible, and better, to tell `except` to catch specific exceptions as in:

```
try:
    int(alphabet)
    int('12.0')
except ValueError:
    print('Caught a value error!')
```

```
except TypeError:
    print('Caught a type error!')
```

that prints:

```
Caught a value error!
```

1.8 Functions

We define a function in Python with the `def` keyword and we use `return` to return the results. In Sect. 1.6.6, we wrote a small program to count the letters of a text. Let us create a function from it that accepts any text instead of `iliad`. We also add a Boolean, `lc`, to set the text in lowercase:

```
def count_letters(text, lc=True):
    letter_count = {}
    if lc:
        text = text.lower()
    for letter in text:
        if letter.lower() in alphabet:
            if letter in letter_count:
                letter_count[letter] += 1
            else:
                letter_count[letter] = 1
    return letter_count
```

We call the function with the two parameters:

```
count_letters(iliad, True)
```

If most of the calls use a parameter with a specific value, we can use it as default with the notation:

```
def count_letters(text, lc=True):
```

In this case, the call `count_letters(iliad)` with one single parameter will be equivalent to:

```
count_letters(iliad, True)
```

1.9 Comprehensions and Generators

1.9.1 Comprehensions

Instead of loops, the comprehensions are an alternative, concise syntactic notation to to create lists, sets, or dictionaries.

An example of elegant use of list comprehensions is given by Norvig (2007), who wrote a delightful spelling corrector in 21 lines of Python. To verify that a word is correctly written, spell checkers look it up in a dictionary. If a word is not in the dictionary, and is presumably a typo, spelling correctors generate candidate corrections through, for instance, the deletion of one character of this word, in the hope that it can find a match in the dictionary. See Sect. **??** of this book for details.

Given an input word, we can generate all the one-character deletions in two steps: First, we split the word into two parts; then we delete the first letter of the second part. We can write this operation in two comprehensions, whose syntax is close to the set comprehension in set theory. First, we generate the splits:

```
splits = [(word[:i], word[i:]) for i in range(len(word) + 1)]
```

where we iterate over the sequence of character indices and we create pairs consisting of a prefix and a rest.

If the input word is *acress*, the resulting list in `splits` is:

```
[(’’, ’acress’), (’a’, ’cress’), (’ac’, ’ress’),
(’acr’, ’ess’), (’acre’, ’ss’), (’acres’, ’s’), (’acress’, ’’)]
```

Then, we apply the deletions, where we concatenate the prefix and the rest deprived from its first character. We check that the rest is not an empty list:

```
deletes = [a + b[1:] for a, b in splits if b]
```

The result in `deletes` is:

```
[’cress’, ’aress’, ’acess’, ’acrss’, ’acres’, ’acres’]
```

where *cress* and *acres* are dictionary words.

The comprehensions are equivalent to loops. The first one to:

```
splits = []
for i in range(len(word) + 1):
    splits.append((word[:i], word[i:]))
```

and the second one:

```
deletes = []
for a, b in splits:
    if b:
        deletes.append(a + b[1:])
```

We can create set and dictionary comprehensions the same way by replacing the enclosing square brackets with curly braces: `{}`.

1.9.2 Generators

List comprehensions are stored in memory. If the list is large, it can exceed the computer capacity. Generators generate the elements on demand instead and can handle much longer sequences.

Generators have a syntax that is identical to the list comprehensions except that we replace the square brackets with parentheses:

```
splits_generator = ((word[:i], word[i:])
                    for i in range(len(word) + 1))
```

We can iterate over this generator exactly as with a list. The statement:

```
for i in splits_generator: print(i)
```

prints

```
('', 'acress')
('a', 'cress')
('ac', 'ress')
('acr', 'ess')
('acre', 'ss')
('acres', 's')
('acress', '')
```

However, this iteration can only be done once. We need to create the generator again to retraverse the sequence.

Finally, we can also use functions to create generators. We replace the `return` keyword with `yield` to do this, as in the function:

```
def splits_generator_function():
    for i in range(len(word) + 1):
        yield (word[:i], word[i:])
```

that returns a generator identical to the previous one:

```
splits_generator = splits_generator_function()
```

1.9.3 Iterators

We just saw that we can iterate only once over a generator. Objects with this property in Python are called iterators. Iterators are very efficient devices and, at the same time, probably less intuitive than lists for beginners.

Let us give some examples with a useful iterator: `zip()`. Let us first create three strings with the Latin, Greek, and Russian Cyrillic alphabets:

```
latin_alphabet = 'abcdefghijklmnopqrstuvwxyz'
len(latin_alphabet)        # 26
greek_alphabet = 'αβγδεζηθικλμνξοπρστυφχψω'
len(greek_alphabet)        # 24
cyrillic_alphabet = 'абвгдеёжзийклмнопрстуфхцчшщъыьэюя'
len(cyrillic_alphabet)     # 33
```

`zip()` weaves strings, lists, or tuples and creates an iterator of tuples, where each tuple contains the elements with the same index: `latin_alphabet[0]` and `greek_alphabet[0]`, `latin_alphabet[1]` and `greek_alphabet[1]`, and so on. If the strings are of different sizes, `zip()` will stop at the shortest.

The following code applies `zip()` to the three first letters of our alphabets:

```
la_gr = zip(latin_alphabet[:3], greek_alphabet[:3])
la_gr_cy = zip(latin_alphabet[:3], greek_alphabet[:3],
               cyrillic_alphabet[0:3])
```

and creates two iterators with the tuples:

```
la_gr # (’a’, ’α’), (’b’, ’β’), (’c’, ’γ’)
la_gr_cy # (’a’, ’α’, ’а’), (’b’, ’β’, ’б’), (’c’, ’γ’, ’в’)
```

Once created, we access the elements of an iterator with `__next()__` as in:

```
la_gr.__next__()   # (’a’, ’α’)
la_gr.__next__()   # (’b’, ’β’)
la_gr.__next__()   # (’c’, ’γ’)
```

When we reach the end and there are no more elements, Python raises an exception:

```
la_gr.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

If we want to use this iterator again, we have to recreate it.

Another way to traverse this sequence multiple times is to convert the iterator to a list as in:

```
la_gr_cy_list = list(la_gr_cy)
la_gr_cy_list
       # [(’a’, ’α’, ’а’), (’b’, ’β’, ’б’), (’c’, ’γ’, ’в’)]
la_gr_cy_list
       # [(’a’, ’α’, ’а’), (’b’, ’β’, ’б’), (’c’, ’γ’, ’в’)]
```

We must be aware that the list conversion runs the iterator through the sequence and if we try to convert `la_gr_cy` a second time, we just get an empty list:

```
la_gr_cy_list = list(la_gr_cy)
la_gr_cy_list  # []
```

To restore the original lists of alphabet, we can use the `zip(*)` inverse function:

```
la_gr_cy = zip(latin_alphabet[:3], greek_alphabet[:3],
                cyrillic_alphabet[0:3])
       # (’a’, ’α’, ’а’), (’b’, ’β’, ’б’), (’c’, ’γ’, ’в’)
zip(*la_gr_cy)
       # (’a’, ’b’, ’c’), (’α’, ’β’, ’γ’), (’а’, ’б’, ’в’)
```

Finally, we can convert lists to iterators using `iter()`.

1.10 Modules

Python comes with a very large set of libraries called modules like, for example, the `math` module that contains a set of mathematical functions. We load a module with the `import` keyword and we use its functions with the module name as a prefix followed by a dot:

```
import math
math.sqrt(2)            # 1.4142135623730951
math.sin(math.pi/2)     # 1.0
math.log(8, 2)          # 3.0
```

We can create an alias name to the modules with the `as` keyword:

```
import statistics as stats
stats.mean([1, 2, 3, 4, 5])   # 3.0
stats.stdev([1, 2, 3, 4, 5])  # 1.5811388300841898
```

Modules are just files, whose names are the module names with the `.py` suffix. To import a file, Python searches first the standard library, the files in the current folder, and then the files in `PYTHONPATH`.

When Python imports a module, it executes its statements just as when we run:

```
$ python module.py
```

If we want to have a different execution when we run the program from the command line and when we import it, we need to include this condition:

```
if __name__ == '__main__':
    print("Running the program")
    # Other statements
else:
    print("Importing the program")
    # Other statements
```

The first member is executed when the program is run from the command line and the second one, when we import it.

1.11 Installing Modules

Python comes with a standard library of modules like `math`. Although comprehensive, we will use external libraries in the next chapters that are not part of the standard release as the `regex` module in Chap. **??**. We can use `pip`, the Python package manager to install the modules we need. `pip` will retrieve them from the Python package index (PyPI) and fetch them for us.

To install `regex`, we just run the command:

```
$ pip install regex
```

or

```
$ python -m pip install regex
```

and if we want to upgrade an already installed module, we run:

```
$ python -m pip install --upgrade regex
```

Another option is to use a Python distribution with pre-installed packages like Anaconda (`https://www.continuum.io/downloads`). Nonetheless, even if Anaconda has many packages, it does not include `regex` and we will have to install it.

1.12 Basic File Input/Output

Python has a set of built-in input/output functions to read and write files: `open()`, `read()`, `write()`, and `close()`.

The next lines open and read the `iliad.txt` file, count the characters, and write the results in the `iliad_stats.txt` file:

```
f_iliad = open('iliad.txt', 'r')         # open a file
iliad_txt = f_iliad.read()               # read all the file
f_iliad.close()                          # close the file
iliad_stats = count_letters(iliad_txt)   # count the letters
with open('iliad_stats.txt', 'w') as f:
    f.write(str(iliad_stats))
    # we automatically close the file
```

where `open()` opens a file in the read-only mode, `r`, and returns a file object; `read()` reads the entire content of the file and returns a string; `close()` closes the file object; `count_letter()` counts the letters; and finally the `with` statement is a shorthand to handle exceptions and close the file automatically after the block: `open()` creates a new file using the write mode, `w`, and `write()` writes the results as a string.

In addition to these base functions, Python has modules to read and write a large variety of file formats.

1.13 Memo Functions and Decorators

1.13.1 Memo Functions

Memo functions are functions that remember a result instead of computing it. This process is also called *memoization*. The Fibonacci series is a case, where memo functions provide a dramatic execution speed up.

The Fibonacci sequence is defined by the relation:

$$F(n) = F(n-1) + F(n-2)$$

with $F(1) = F(2) = 1$.

A naïve implementation in Python is straightforward:

```
def fibonacci(n):
    if n == 1: return 1
    elif n == 2: return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

however, this function has an expensive double recursion that we can drastically improve by storing the results in a dictionary. This store, `f_numbers`, will save an exponential number of recalculations:

```
f_numbers = {}

def fibonacci2(n):
    if n == 1: return 1
    elif n == 2: return 1
    elif n in f_numbers:
        return f_numbers[n]
    else:
        f_numbers[n] = fibonacci2(n - 1) + fibonacci2(n - 2)
        return f_numbers[n]
```

1.13.2 Decorators

Python decorators are syntactic notations to simplify the writing of memo functions (they can be used for other purposes too).

Decorators need a generic memo function to cache the results already computed. Let us define it:

```
def memo_function(f):
    cache = {}

    def memo(x):
        if x in cache:
            return cache[x]
        else:
            cache[x] = f(x)
            return cache[x]

    return memo
```

Using this memo function, we can redefine `fibonacci()` with the statement:

```
fibonacci = memo_function(fibonacci)
```

that results in `memo()` being assigned to the `fibonacci()` function. When we call `fibonacci()`, we in fact call `memo()` that will either lookup the cache or call the original `fibonacci()` function.

One detail may be puzzling: How does the new function know of the `cache()` variable and its initialization as well as the value of the `f` argument, the original `fibonacci()` function? This is because Python implements a closure mechanism that gives the inner functions access to the local variables of their enclosing function.

Now the decorators: Python provides a short notation for memo functions; instead of writing:

```
fibonacci = memo_function(fibonacci)
```

we just decorate `fibonacci()` with the `@memo_function` line before it:

```
@memo_function
def fibonacci(n):
...
```

1.14 Object-Oriented Programming

Although not obvious at first sight, Python is an object-oriented language, where all the language entities are objects inheriting from a class: The `str` class for the strings, for instance. Each class has a set of methods that we call with the `object.method()` notation.

1.14.1 Classes and Objects

We define our own classes with the `class` keyword. In Sect. 1.8, we wrote a `count_letters()` function that basically is to be applied to a text. Let us reflect this with a `Text` class and let us encapsulate this function as a method in this class. In addition, we give the `Text` class four variables: The content, its length, and the letter counts, which will be specific to each object and the `alphabet` string that will be shared by all the objects. We say that `alphabet` is a class variable while `content`, `length`, and `letter_count` are instance variables.

We encapsulate a function by inserting it as a block inside the class. Among the methods, one of them, the **constructor**, is called at the creation of an object. It has the `__init()__` name. This notation in Python is, unfortunately, not as intuitive as the rest of the language, and we need to add a `self` extra-parameter to the methods as well as to the instance variables. This `self` keyword denotes the object itself. We use `__init()__` to assign an initial value to the `content`, `length`, and `letter_count` variables.

Finally, we have the class:

```
class Text:
    """Text class to hold and process text"""

    alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

```
    def __init__(self, text=None):
        """The constructor called when an object
        is created"""

        self.content = text
        self.length = len(text)
        self.letter_counts = {}

    def count_letters(self, lc=True):
        """Function to count the letters of a text"""

        letter_counts = {}
        if lc:
            text = self.content.lower()
        else:
            text = self.content
        for letter in text:
            if letter.lower() in self.alphabet:
                if letter in letter_counts:
                    letter_counts[letter] += 1
                else:
                    letter_counts[letter] = 1
        self.letter_counts = letter_counts
        return letter_counts
```

We create new objects using the `Text(init_value)` syntax:

```
txt = Text("""Tell me, O Muse, of that many-sided hero who
traveled far and wide after he had sacked the famous town
of Troy.""")
```

A class has its own type:

```
type(txt)           # <class '__main__.Text'>
```

We access the instance variables using this notation:

```
text.length         # 111
```

We create and assign new instance variables the same way:

```
txt.my_var = 'a'    # a new instance variable with value 'a'
txt.content = open('iliad.txt', 'r').read()
                    # txt.content is now the content of the file
```

and we call methods with the same notation:

```
txt.count_letters() # return the letter counts of txt.text
```

Finally, we added short descriptions of the class and its methods in the form of **docstrings**: Strings being the first statement of the class, method, or function. Docstrings are very useful to document a program. We access them using the `.__doc__` variable as in:

```
Text.__doc__    # 'Text class to hold and process text'
Text.count_letters.__doc__
                # 'Function to count the letters of a text'
```

or with the `help()` function.

1.14.2 Subclassing

Using classes, we can build a hierarchy, where the subclasses will inherit methods from their superclass parents.

Let us create a `Word` class that we define as a subclass of `Text`. As we have seen in Sect. **??**, each word has a part of speech, a category, such as verb, noun, pronoun, adjective, etc. Let us add this part of speech as an instance variable `part_of_speech` and let us add an `annotate()` function to assign a word with its part of speech. We have the new class:

```
class Word(Text):
    def __init__(self, word=None):
        super().__init__(word)
        self.part_of_speech = None

    def annotate(self, part_of_speech):
        self.part_of_speech = part_of_speech
```

where the `super().__init__(word)` function will call the constructor of `Text`.

We can then create a new word:

```
word = Word('Muse')
```

that inherits the `Text` instance variables:

```
word.length         # 4
```

and methods

```
word.count_letters(lc=False))
                    # {'M': 1, 'u': 1, 's': 1, 'e': 1}
```

We can also call the `Word` specific method as:

```
word.annotate('Noun')
```

and have:

```
word.part_of_speech # Noun
```

1.15 Functional Programming

Python provides some functional programming mechanisms with map and reduce functions.

1.15.1 map()

map() enables us to apply a function to all the elements of an iterable, a list for instance. The first argument of map() is the function to apply and the second one, the iterable. map() returns an iterator.

Let us use map() to compute the length of a sequence of texts, in our case, the first sentences of the *Iliad* and the *Odyssey*. We apply len() to the list of strings and we convert the resulting iterator to a list to print it.

```
odyssey = """Tell me, O Muse, of that many-sided hero who
traveled far and wide after he had sacked the famous town
of Troy."""

text_lengths = map(len, [iliad, odyssey])
list(text_lengths)     # [100, 111]
```

1.15.2 Lambda Expressions

Let us now suppose that we have a list of files instead of strings, here iliad.txt and odyssey.txt. To deal with this list, we can replace len() in map() with a function that reads a file and computes its length:

```
def file_length(file):
    return len(open(file).read())
```

For such a short function, a lambda expression can do the job more compactly. A lambda is an anonymous function, denoted with the lambda keyword, followed by the function parameters, a colon, and the returned expression. To compute the length of a file, we write the lambda:

```
lambda file: len(open(file).read())
```

and we apply it to our list of files:

```
files = ['iliad.txt', 'odyssey.txt']
text_lengths = map(lambda x: len(open(x).read()), files)
list(text_lengths)                  # [809768, 611742]
```

We can return multiple values using tuples. If we want to both keep the text and its length in the form of a pair: $(text, length)$, we just write:

```
text_lengths = (
    map(lambda x: (open(x).read(), len(open(x).read())),
        files))
text_lengths = list(text_lengths)
[text_lengths[0][1], text_lengths[1][1]]  # [809768, 611742]
```

In the previous piece of code, we had to read the text twice: In the first element of the pair and in the second one. We can use two `map()` calls instead: One to read the files and a second to compute the lengths. This results in:

```
text_lengths = (
    map(lambda x: (x, len(x)),
        map(lambda x: open(x).read(), files)))
text_lengths = list(text_lengths)
[text_lengths[0][1], text_lengths[1][1]]  # [809768, 611742]
```

1.15.3 `reduce()`

`reduce()` is a complement to `map()` that applies an operation to pairs of elements of a sequence. We can use `reduce()` and the addition to compute the total number of characters of our set of files. We formulate it as a lambda expression:

```
lambda x, y: x[1] + y[1]
```

to sum the consecutive elements, where the length of each file is the second element in the pair; the first one being the text.

`reduce()` is part of the `functools` module and we have to import it. The resulting code is:

```
import functools

char_count = functools.reduce(
    lambda x, y: x[1] + y[1],
    map(lambda x: (x, len(x)),
        map(lambda x: open(x).read(), files)))
char_count     # 1421510
```

1.15.4 `filter()`

`filter()` is a third function that we can use to keep the elements of an iterable that satisfy a condition. `filter()` has two arguments: A function, possibly a lambda, and an iterable. It returns the elements of the iterable for which the function is true.

As an example of the `filter()` function, let us write a piece of code to extract and count the lowercase vowels of a text.

We need first a lambda that returns true if a character `x` is a vowel:

```
lambda x : x in 'aeiou'
```

that we apply to the iliad string to obtain all its vowels:

```
''.join(filter(lambda x : x in 'aeiou', iliad))
     # ioeeaeoieooeeuaououeiuoeaea
```

We can apply the same code to a whole file:

```
''.join(filter(lambda x: x in 'aeiou',
               open('iliad.txt').read()))
```

and easily extend the extraction to a list of files using `map()`:

```
map(lambda y:
    ''.join(filter(lambda x: x in 'aeiou',
                   open(y).read())),
    files)
```

We finally count the vowels in the two files using `len()` that we apply with a second `map()`:

```
list(map(len,
         map(lambda y:
             ''.join(filter(lambda x: x in 'aeiou',
                            open(y).read())),
             files)))  # [231874, 176190]
```

1.16 Further Reading

Python has become very popular and there are plenty of good books or tutorials to complement this introduction. `Python.org` is the official site of the Python software foundation, where one can find the latest Python releases, documentation, tutorials, news, etc. It also contains masses of pointers to Python resources. Anaconda is an alternative Python distribution that includes many libraries (`www.continuum.io/downloads`).

Python comes with a integrated development environment (IDE) called IDLE that fulfills basic needs. PyCharm is a more elaborate code editor with a beautiful interface. It has a free community edition (`www.jetbrains.com/pycharm/`). IPython is an interactive computing platform, where the programmer can mix code and text in the form of notebooks.

Among all the computer science publishers, O'Reilly Media has an impressive collection of books on Python that ranges from introductions to very detailed or domain-oriented monographs. Payne (2015) and Lutz (2013) are two examples of this variety.

Finally, online course providers, like Coursera or edX, offer free and high-quality Python courses from top universities and open to anyone.

Index

References

Guo, P. (2014). Python is now the most popular introductory teaching language at top U.S. universities. `http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universiti` `fulltext`. Retrieved November 23, 2015.

Lutz, M. (2013). *Learning Python*. O'Reilly Media, Sebastopol, California, 5th edition.

Norvig, P. (2007). How to write a spelling corrector. `http://norvig.com/spell-correct.html`. Cited 30 November 2015.

Payne, B. (2015). *Teach Your Kids to Code: A Parent-Friendly Guide to Python Programming*. No Starch Press, San Francisco.

van Rossum, G., Warsaw, B., and Coghlan, N. (2013). PEP 0008 – Style guide for Python code. `https://www.python.org/dev/peps/pep-0008/`. Retrieved November 23, 2015.