Basic programming

Emiel van Miltenburg

August 15, 2019

Preface

This book grew out of the *Python for text analysis* course that I co-taught at the *Vrije Universiteit Amsterdam* with Marten Postma. Before we started teaching the course, I wanted to get a good overview of what the course should be about, and I wrote the first version of this text. We never really used this text, except for answering Frequently Asked Questions. The main body of the course was (and still is) a collection of Jupyter Notebooks. The course developed over the years, with contributions from Chantal van Son, Filip Ilievski, and Pia Sommerauer. It is still available at: `https://github.com/cltl/python-for-text-analysis`.

Later, at Tilburg University, I wanted to have a reference text to serve as a reference for students in Communication and Information Science, so I decided to use my old 'brain dump' of a text as a basis for an introductory book on Python programming. At the moment, it is still an early draft, but I hope this text provides a useful summary next to the weekly exercises in our course. Any feedback is welcome; just send me a message via: C.W.J.vanMiltenburg@uvt.nl

Contents

Chapter 1

The basics

Welcome to our Python course! This first chapter covers a lot of material. You do not need to know everything by heart just yet. This page is intended as a global reference that you will need less and less as we progress through the course.

- If you are the kind of person who likes to have a high-level overview of the materials for each week, then start with these documents first (we will have some theory every week).
- If you are the kind of person who likes to get some hands-on experience first, and then reflect on the theory afterwards, then start with the notebook.

1.1 Python: an object-oriented language

Object-oriented programming means that you think about programming in terms of manipulating different kinds (or *classes*) of objects. These are the classes that Python has built-in:

- **String** for text.
- **Integers** and **floats** for whole and decimal numbers.
- **Booleans** for truth values.
- **Lists** for ordered sequences of objects.
- **None** for the special value of 'nothing'.
- **Set** and **frozenset** for unordered collections of objects.
- **Dictionary** for mappings between objects.
- **Tuple** for immutable ordered sequences.

This distinction is useful, because the creators of Python have built in many handy class-specific *methods* (ways to manipulate a particular object). So when you enter a string like `"Dog"` or `"This is a sentence."`, the Python interpreter immediately knows what you can do with it. For example: you can compute the sum of two numbers, or count the number of letters in a word.

An example

Let's take an example with strings. The box below shows code I typed into my Python interpreter. >>> indicates input. The output is shown without >>>.

```
>>> text = 'Here are some words'
>>> print(text)
Here are some words
>>> type(text)
<class 'str'>
```

What happened here?

1. First we **assigned** the **value** `'Here are some words'` to the **variable** `text`.
2. Then we issued the `print`-function to print the value of `text`. (Printing is just outputting some value to the screen.) As the output, we got `Here are some words` without the single quotes. The single quotes were just there to tell Python that all the characters in `'Here are some words'` belong together as a single string.
3. Then we asked Python about the **type** of the variable `text`, and the interpreter answered that `text` is an **instance** of the class `'str'` (Pythons name for strings).

1.2 Methods

OK, so now we know that `text` corresponds to an instance of the string-class, with the value `'Here are some words'`.The Python-documentation has an overview of all the *methods* you can use with strings. Methods provide the most common operations that you can perform with a particular class. For example, there is a `startswith()` method that we can use to see if our text starts with the word 'here':

```
>>> text.startswith('Here')
True
>>> text.startswith('Her') # Only characters matter. Not whether it's a word.
True
>>> text.startswith('Floppy')
False
>>> text.startswith('her') # But the method IS case-sensitive.
False
```

You can call any method using the dot-notation exemplified above. Because methods are defined per class, you cannot use a string-method with an integer. If you try to do so, the Python interpreter will complain:

```
>>> x = 11
>>> x.startswith(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'startswith'
```

If you really do want to perform string operations on an integer, the only thing you can do is **cast** the variable to another type using the built-in `str()` function:

```
>>> x = str(x)
>>> x.startswith('1')
True
```

What happened here is that the **integer** 11 is re-interpreted as a **string** of two characters: '1' and '1'. For other built-in type methods, see the built-in functions section below.

1.3 Lists

Explain how lists work.

1.4 Numerical types: floats and integers

1.4.1 Floats and integers

1.4.2 Mathematical operators

If you want to work with numbers, you will need to use mathematical operators. These are defined for both **integers** and **floats** (numeric types):

Operation	Result
`x + y`	sum of x and y
`x - y`	difference of x and y
`x * y`	product of x and y
`x / y`	quotient of x and y
`x // y`	floored quotient of x and y
`x % y`	remainder of x / y
`-x`	x negated
`+x`	x unchanged
`x ** y`	x to the power y

But two of these operators (+ and *) also work for other types of objects. E.g. for lists:

```
# Addition:
>>> [1] + [2]
[1, 2]
# Multiplication:
>>> [1] *8
[1, 1, 1, 1, 1, 1, 1, 1]
# Combined:
>>> [1] * 4 + [2] * 4
[1, 1, 1, 1, 2, 2, 2, 2]
```

Or for strings:

```
>>> 'a' + 'b'
'ab'
>>> 'a' * 4
'aaaa'
>>> 'a' * 4 + 'b' * 4
'aaaabbbb'
```

1.5 Sets, lists, and tuples

Sets and tuples are often confusing to people just starting out with Python; if there are already lists, why bother with these other container types? But after working with Python for a while, you'll start to see that each of them has its own strengths and weaknesses, and they're all useful in different contexts.

- Lists are useful because they are ordered and indexable. You can also easily add stuff to the head or the tail of a list. Importantly, the same object can occur multiple times in the same list.
- Sets are useful because they are fast. If you want to check whether an object is in a container, use sets. (For lists, the interpreter needs to check every element, while element lookup for a set is instantaneous, because all its elements are hashed –see the note on mutability below.) Items cannot occur more than once in a list, and duplicates are removed. This is perfect for storing vocabularies!
- Tuples are useful because they are immutable and hashable. Read the note below for what that means exactly, but in practice you'll find that tuples are very useful as objects to return from a function, and as keys to dictionaries.

Note that you can switch between types (i.e. *casting*) at all times using the type-related built-in functions (see further below).

Side-note: Mutability

We will sometimes mention that some object is (im)mutable. The Python glossary provides useful definitions:

- **Immutable**: An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.
- **Mutable**: Mutable objects can change their value but keep their id(). See also immutable.

Examples of mutable objects are lists, sets, and dictionaries. Why this is relevant is a topic for a later week. If you area really interested, Ned Batchelder has an excellent talk about Python names and values that discusses these concepts. Objects that are instances of a built-in immutable type are *hashable* and can be used as keys in a dictionary. For a really good explanation of this, see the YouTube video the mighty dictionary.

1.6 Dictionaries

Dictionaries are known in other languages under various names, such as 'hash tables', or 'maps'. You can use them to associate **keys** and **values**. The easiest example is a shopping list:

```
>>> shopping_list = {'bacon': 2, 'eggs': 6, 'spam':50}
>>> print(shopping_list.keys())
dict_keys(['spam', 'bacon', 'eggs'])
```

```
# How any eggs should I buy?
>>> print(shopping_list['eggs'])
6
```

Dictionaries have one limitation: you can only use hashable objects as keys. If you haven't already, see the YouTube video the mighty dictionary to find out how dictionaries work exactly. The video is a bit older, and Python further developed in the mean time (main change: all dictionaries are ordered by default in Python 3), but the explanation of hashing is still relevant. Curtis Lassam's Hash Functions and You: Partners in Freedom also provides a good explanation of hashing.

1.7 More methods

- List methods are given here.
- Dictionary methods are given here
- Set methods are given here

But if you don't want to look these up, you can always use a combination of the `dir()` and `help()` functions that are built into Python. The former gives you a list of methods for the variable that you call the function with:

```
>>> dir(text)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mod__','__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold',
'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

The double underscore ('dunder') methods are Python-internal. We will cover some of them later on in this course. If you want to know what any of these methods do, you can just use the 'help()' function on them, like so:

```
>>> help(text.split)
```

The result of this operation is an explanation that looks like this:

```
Help on built-in function split:

split(...) method of builtins.str instance
    S.split(sep=None, maxsplit=-1) -> list of strings

    Return a list of the words in S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator and empty strings are
    removed from the result.
```

If you are doing this on the command line, you might have entered a different 'mode' to read the help text. Press 'q' and <enter> to go back to the interpreter.

1.8 Working with sequences

Lists, strings, and tuples are all ordered sequences. The `range()`-object is also a sequence. Sequences support a series of common operations, copied here for convenience (omitting the notes):

Operation	Result
`x in s`	`True` if an item of s is equal to `x`, else `False`
`x not in s`	`False` if an item of s is equal to `x`, else `True`
`s + t`	the concatenation of `s` and `t`
`s * n or n * s`	equivalent to adding `s` to itself `n` times
`s[i]`	`i`th item of `s`, origin 0
`s[i:j]`	slice of `s` from `i` to `j`
`s[i:j:k]`	slice of `s` from `i` to `j` with step `k`
`len(s)`	length of `s`
`min(s)`	smallest item of `s`
`max(s)`	largest item of `s`
`s.index(x[, i[, j]])`	index of the first occurrence of `x` in `s` (at or after index `i` and before index `j`)
`s.count(x)`	total number of occurrences of `x` in `s`

All sequences are *iterable*, which means that you can iterate over them using a for-loop. Here is a small example of a for-loop:

```python
fruits = ['apple', 'pear', 'orange']
for fruit in fruits:
    print(fruit)

# You can also do this in different orders, using the built-in
# functions sorted() and reversed():
for fruit in sorted(fruits):
    print(fruit)

for fruit in reversed(fruits):
    print(fruit)
```

This for-loop iterates over the list of fruits. Each round the variable `fruit` gets updated so that it refers to the next string in the list. Then it prints the relevant string and continues with the next one, until all strings have been printed.

Other things that are iterable include dicts, sets, and *file objects*. (If you iterate over a dictionary you get keys, and if you iterate over a file you get lines.) But because sets and dictionaries are not ordered, you cannot index them (i.e. use square brackets, like: `s[i]`). Ned Batchelder has a great video about iteration.

1.9 Built-in functions

In addition to the class methods, Python also provides a set of general, built-in functions that you should know by heart. See this section of the library reference. We'll mostly cover the relevant ones in the notebooks, but here's a selection with short descriptions:

1.9.1 Type-related

- `bool()` turns an object into a boolean (True or False).
- `dict()` without an argument, it creates an empty dictionary, but can also be used with arguments to populate the dictionary. See the notebook for exercises.
- `float()` Turns an integer or a string-representation of a floating point number into a floating point number.
- `int()` Turns a float or a string-representation of a natural number into an integer.
- `list()` Creates an empty list or turns an iterable into a list.
- `set()` Creates an empty set or turns an iterable into a set.
- `str()` Turns objects into strings.
- `tuple()` Turns iterables into tuples.
- `type()` can be used to check the type of a variable.

1.9.2 Math-related

- `sum(iterable)` returns the sum of an iterable.
- `divmod(x,y)` takes two ints and returns the quotient and the remainder.
- `pow(x,y)` raises `x` to the power of `y`.
- `round(x[,n])` rounds its argument to the nearest integer, or the nearest float with `n` digits.
- `min(iterable)` returns the lowest number in an iterable. (This function can also take any number of arguments to do the same for those arguments.)
- `max(iterable)` returns the highest number in an iterable. (This function can also take any number of arguments to do the same for those arguments.)

Methods and built-in functions are the first places to check if you want to solve a programming question. Ask yourself: *What kind of data am I working with?* If it's text data, check out the string methods and the string-related modules. If it's numeric data, check out the built-in functions and operators, and the `math` module. We will talk about modules soon.

1.10 Later on in the course

We will also see that you can define new classes, using either the general **object** as a base class, or using one of the classes mentioned above as a base class. This means that you can add more functionality to existing classes. Examples from the standard library are `Counter` and `defaultdict` in the `collections` module, that use `dict` as their base class.

1.11 Useful links

Here are some links that may be useful to you.

- A whirlwind tour of Python
- General documentation
- Stack Overflow
- Learning resources on /r/learnpython

If you take a piece of code from StackOverflow (or some other website), always acknowledge the source in the comments, and explain what that bit of code does. (This is also to protect yourself: you should never run code that you haven't verified or don't understand.)

1.11.1 GitHub and Markdown

We will talk about GitHub and Markdown in the second part of this course.

Move to other part of this document.

- GitHub Desktop
- GitHub Tutorial
- Markdown

Chapter 2

Control flow tools and files

This chapter we will start working with files, and introduce **for-loops, while-loops**, and **functions**. You will learn how to define your own functions, and use those functions to analyze text. We will also discuss different file formats.

2.1 Conditionals: if, elif, else

`if...elif...else...`-statements provide a powerful way to structure your code. You can use them to run different bits of code depending on a particular set of conditions. Here is an example:

```
# Inter-rater reliability is a measure to assess annotation quality.
# Here is some code to interpret Cohen’s Kappa.
# (according to Landis & Koch 1977)

if kappa == 0:
    print(’Poor’)
elif kappa <= 0.2:
    print(’Slight’)
elif kappa <= 0.4:
    print(’Fair’)
elif kappa <= 0.6:
    print(’Moderate’)
elif kappa <= 0.8:
    print(’Substantial’)
else:
    print(’(Almost) perfect’)
```

While `if...elif...else...`-statements are very useful, they do not always provide the optimal solution. Here is one example:

```
# Beware if you do something like this:
if x == 1:
    y = ’a’
elif x == 2:
    y = ’b’
elif x == 3:
    y == ’c’
elif x == 4:
    y = 'd'

# Use a dictionary instead:
d = {1:'a', 2:'b', 3:'c', 4:'d'}
y = d[x]
```

The reason why the conditionals worked so well in the first example is that they cover a range of values. This is almost impossible to capture in a dictionary.

Most often, you'll probably use a single 'if'-statement, or an 'if' combined with an 'else'. Long sequences of `if...elif...else...` like in the first example are not that common, but it's very useful to know!

2.2 For-loops

There are two kinds of loops: the for-loop and the while-loop. This paragraph first introduces the for-loop, which is the most commonly used loop in Python. You'll find that, most of the time, you just want to carry out some operation for all the items in a sequence. And that's just what the for-loop does! It looks like this:

```
for number in [1,2,3]:
    print(number)
```

This loop prints the numbers 1, 2, and 3, each on a new line. The variable name `number` is just something I have chosen. It could have been anything, even something like `sugar_bunny`. But `number` is nice and descriptive. OK, so how does the loop work?

1. The Python interpreter starts by checking whether there's anything to iterate over. If the list is empty, it just passes over the for-loop and does nothing.
2. Then, the first value in the iterable (in this case a list) gets assigned to the variable `number`.
3. Following this, we enter a 'local context', indicated by the indentation (four spaces preceding the print function). This local context can be as big as you want. All Python cares about is those four spaces. Everything that is indented is part of the local context.
4. Then, Python carries out all the operations in the local context. In this case, this is just `print(number)`. Because `number` refers to the first element of the list, it prints `1`.
5. Once all operations in the local context have been carried out, the interpreter checks if there are any more elements in the list. If so, the next value (in this case `2`) gets assigned to the variable `number`.
6. Then, we move to step 3 again: enter the local context, carry out all the operations, and check if there's another element in the list, and so on, until there are no more elements left.

2.2.1 Ranges of numbers

Sometimes, it is useful to do the same thing a number of times. To do this, you can use the `range()` function:

```
for i in range(10):
    print('I like repeating myself!')
```

This will print 'I like repeating myself' ten times. `range()` is a function that returns a `range` object. Looping over that object will give you all the numbers in a particular range, e.g. `range(5,9)` corresponds to [5,6,7,8], excluding 9.

(Note that `range` is not a list. It only keeps one number in memory at any given time, so it's really memory-efficient.)

2.2.2 Enumerating sequences

Sometimes, it is also useful to have the index of an item as you iterate over a list. For this purpose, there is another useful built-in function called `enumerate()`.. It works like this:

```
for index, character in enumerate('Monty'):
    print('character with index', index, 'is', character)
```

The code above prints:

```
character with index 0 is M
character with index 1 is o
character with index 2 is n
character with index 3 is t
character with index 4 is y
```

Note that we make use of **multiple assignment** here. Multiple assignment is the practice of assigning values to multiple variables at once. The simplest example of multiple assignment is this:

```
>>> x,y = (1,2)
>>> print(x)
1
>>> print(y)
2
```

The `enumerate` example above is equivalent to this:

```
for pair in enumerate('Monty'):
    index, character = pair
    print('character with index', index, 'is', character)
```

Or even this:

```
for pair in enumerate('Monty'):
    index = pair[0]
    character = pair[1]
    print('character with index', index, 'is', character)
```

But both are longer and not as clear. We'll talk more about `enumerate` and multiple assignment in the notebook.

2.3 While-loops

While-loops are the oldest type of loops. Every programming language that has loops, has a while-loop. Here is an example:

```
i = 0
while i < 10:
    print(i)
    i += 1
```

This is equivalent to:

```
for i in range(10):
    print(i)
```

While-loops always start with the word `while`, followed by a **(boolean) condition**. You can read the first line of a while-loop as: "while the condition holds, perform all the operations in the local context." In this case, the only two operations are `print(i)` and `i += 1`. This incrementation of `i` ensures that the loop will eventually end, once `i` is equal to 10.

As you can see above, the `for`-loop is much shorter than the `while`-loop. This is usually the case when all you want to do is iterate over some collection of things. Since 'iterating over a collection of things' is usually enough for our purposes, we will mainly use the for-loop. So when do you use `while`? Our rule-of-thumb is this:

- Use `while` if there is a clear condition for success (or failure), and you're not sure how long it will take to get to that point.

For example:

- If you're collecting data from webpage that doesnt always load: keep trying until it loads.
- If you're mining web pages for information, and you want to collect a certain amount of that information: keep looking until you have enough.
- If you're working with a Queue or a to-do list: keep working until there is nothing more to do. (You can even add stuff to the list during the loop.)

2.4 Functions

A function is really just a convenient way to re-use code. We've actually already seen several kinds of functions. For example:

- The print function. All this function does is take some input (any object), and display that input on the screen.
- The `min()` and `max()` functions. These take a collection, and **return** either the smallest or the largest element of that collection.

The word 'return' is used when a function produces any output that can be used for further computation. For example: `min([1,2,3])` returns `1`. But `print('hello')` does not return anything. It just outputs text on your screen. When a function returns output, you can assign that output to a variable, like so:

```
x = min([1,2,3])
print(x) # This will print '1'.
```

When you try to do the same with ‘print(’hello’)‘, you get a different result:

```
x = print('hello')
print(x) # This will print None.
```

You could write a simple implementation of `min()` every time you wanted to get the smallest number. For example:

```
list_of_numbers = [2,3,1,4,5,6,2,4,0]
smallest = list_of_numbers[0]
for number in list_of_numbers:
    if number < smallest:
        smallest = number
# To show that this works:
print('smallest number is', smallest)
```

This would print ‘0‘, which is the smallest number in the list. But if you wanted to do this multiple times in your program, it would be a waste of time to write the same piece of code multiple times. Functions are nothing but names for pieces of code that are defined elsewhere. In the definition of `min()`, it is specified that the **argument** of `min()` should correspond to `list_of_numbers`, and that it should output `smallest` after determining which number is the smallest.

2.4.1 Writing your own functions

Here is how you define a function:

- write `def`;
- the name you would like to call your function;
- a set of parentheses containing the argument(s) of your function;
- a colon;
- a **docstring** describing what your function does;
- the function definition;
- ending with a **return statement**.

Here’s a re-definition of the `min()` function that’s already provided with Python:

```
def min_clone(list_of_numbers):
    """
    Function to determine which number is the smallest.
    The input is a list of number, and the output is a specific number.
    """
    smallest = list_of_numbers[0]
    for number in list_of_numbers:
        if number < smallest:
            smallest = number
    return smallest

# To show that this works:
```

```
lon = [2,3,1,4,5,6,2,4,0]
x = min(lon)
print('smallest number is', x)
```

Before you define a function, however, you should always check if that function hasn't been implemented already in Python's standard library. Also: *never* use a function name that has been defined already, even if you mean to replace Python's built-in functionality. (This makes your code more re-usable and future-proof.)

2.5 Working with files

There are two ways to work with files. The preferred way is to use `with`-statement. like this:

```
# Open the file, call it 'f'
with open('some_file.txt') as f:
    # Get the text data.
    text = f.read()
```

Which is equivalent to:

```
# Open the file.
f = open('some_file.txt')
# Get the text data.
text = f.read()
# ...Some time later, close the file.
f.close()
```

In both cases, you create a **file object** called `f` (the conventional name, but you could call it `mickey_mouse` and it would still work). That object has a method called `read()` that returns all the text in the file as one big string. This string value gets assigned to the variable called `text`.

The main advantage of using the `with`-statement is that it automatically closes the file once you leave the local context defined by the indentation level. If you 'manually' open and close the file, you risk forgetting to close the file.

The `open` function can be used in different **modes**. By default it opens files in **read mode**, which means that Python can access the contents, but cannot modify the file. You can make the mode explicit by adding an additional argument. For example, you could use `open('some_file.txt', 'r')` to explicitly state that you want to open the file in read mode. Here is a table with all the modes (copied from here):

2.5.1 Common file operations

Common operations are:

- Skipping a line. For example if the first line of the document doesn't contain relevant information, use: `next(f)`.
- Going over a file line by line. Use a for-loop: `for line in f: ...`. This is strictly preferred over `f.readlines()`!
- Reading all text in a file: `f.read()`
- Writing a single line: `f.write(line)`

Character	Meaning
`r`	open for reading (default)
`w`	open for writing, truncating the file first
`x`	open for exclusive creation, failing if the file already exists
`a`	open for writing, appending to the end of the file if it exists
`b`	binary mode
`t`	text mode (default)
`+`	open a disk file for updating (reading and writing)
`U`	universal newlines mode (deprecated)

- Writing multiple lines: `f.writelines(list_of_lines)`. Note that lines have to end with a newline character (
`n`) or else there won't be any breaks!

We'll practice with these in the notebook.

2.5.2 File names

You can put a lot of information inside a file name. Here's a small list of ideas that may come in useful. Keep in mind how you might retrieve this information from the file name once you've generated it. For example by separating all parts with an underscore, so that you can 'dissect' the file name using the `str.split('_')` operation.

- The date and time. For now you can just assume we'll generate one file, and you can hard-code the date, using the YYYYMMDD format (this format is easiest to sort). Later you can use the time module for this.
- The index/rank of the file in a sequence. You can either include the index in the file name directly, or better: pad the number with zeroes so that the file names are easier to sort and the numbers are nicely aligned: you can do this using the `rjust` command, like this: `'1'.rjust(5,'0')`.
- The extension. Your file doesn't have to end with `.txt` or `.csv`. You can use any extension you like! E.g. '.results' or '.log'.

2.6 File formats

Computers work with many different kinds of files. Different file formats may be useful for different purposes. Below we'll look at some of the most common ones.

2.6.1 Plain text

Plain text is the most basic file format: it's just everyday characters that you're used to, plus some special characters to add whitespace. You will mostly be using tab (`\t`) and newline(`\n`). The only issue with plain text is that the characters maybe stored in some exotic format. In that case you need to convert their encoding to unicode. You can use the codecs module for this.

To learn more about unicode and character sets, see the following **readings**. These two links are super relevant if you want to learn more about proper text handling. Don't worry if you don't understand everything. We will discuss them in class.

- The absolute minimum every software developer absolutely, positively should know about unicode and character sets (no excuses)
- Ned Batchelder on 'pragmatic unicode'

During the course, we will mostly just work with unicode, so encodings won't come up very frequently.

2.6.2 CSV and TSV files

CSV and TSV are one step up from plain text. These formats are used for data that is structured in rows and columns (like a spreadsheet in Excel). Each line corresponds to a row, and cells are either separated by commas (CSV) or tabs (TSV). These files may start with a **header**: a row with labels for each column. We will use the `csv` module to work with CSV and TSV files. See the notebook for exercises.

```
import csv

# Open a TSV file for reading
# For CSV files you can leave out the delimiter argument.
with open('example.tsv') as f:
    reader = csv.reader(f, delimiter='\t')
    for row in reader:
        # Do stuff

# Open a TSV file for writing
with open('example.tsv','w') as f:
    writer = csv.writer(f, delimiter='\t')
    # If rows is an iterable containing rows, you can use writerows()
    # rows could be something like [[1,2,3],[4,5,6]]
    writer.writerows(rows)
```

2.6.3 JSON files

JSON is one of the standard data formats for the web. Wikipedia has a very good description of what JSON files are and what they look like. We will use the `json` module to deal with them.

```
import json

# Load a json file as a dictionary (common use case)
with open('example.json') as f:
    d = json.load(f)

# Write out a dictionary as a JSON file. Note that not all objects can be written
# to a JSON file. E.g. sets are disallowed (you could cast them to lists instead).
with open('example.json','w') as f:
    json.dump(d,f)
```

2.6.4 Making files both human- and machine-readable

There are no strict rules to make files both human- and machine-readable. But there are some principles that you should follow:

- Structure your file.
- Use special characters to separate different parts. For example:
  - Use a colon for text fields (`Name: John`)
  - Use dashes to separate entries (`---------`).
  - Use blank lines to separate different parts of an entry.
- Be consistent in your format.
- Think about usability: people make less mistakes if the format feels natural to use.

We'll work with several different examples in the notebook. One of these is the `linguist list` data, which consists of messages sent to a mailing list. Messages for each day are bundled together and sent to the subscribers of the mailing list. The structure of each set of bundled messages is such that you can recover the main properties of the individual messages. But at the same time, because the mailing list is meant to be read by researchers, the editors took care to present everything in a readable format.

2.6.5 XML and HTML

Finally, we will talk about XML and HTML files. (Click on those links for an explanation of these file types.) These formats are commonly used to store data, or to present it on the web. We will use the `lxml` module to deal with the former, and the `beautifulsoup` library to deal with the latter. See the notebook for examples.

Chapter 3

Batteries included

Python is sometimes described as a 'batteries-included' language. What people mean by this is that the language comes with a rich collection of modules that you can use to solve everyday tasks. There's no need to write any specialized code to do common tasks: others have probably solved the task for you and wrote an efficient implementation. This chapter is devoted to the many modules that come with Python.

Much of what we'll cover below is also in the Python library reference. Below is a list of sections in the library reference we think are most important for our purposes. The first four are full sections that cover the same material as chapters 1 and 2. You may want to read these first, but you can also skip them and move to the built-ins. (They're great reference material, though!)

- **1.** Introduction
- **2.** Built-in Functions
- **3.** Built-in Constants
- **4.** Built-in Types

We will focus on the following sections, making a selection of the most important parts. Of course you can also just read the full documentation if you prefer.

- **6.1** string
- **6.2.** re
- **8.1.** datetime
- **8.3.** collections
- **9.2.** math
- **9.6.** random
- **10.1.** itertools
- **10.2.** functools
- **11.2.** os.path
- **11.7.** glob
- **12.1.** pickle
- **13.2.** gzip
- **14.1.** csv
- **19.2.** json
- **26.3** doctest

3.1 Importing modules

Whenever you want to use a module, you first have to import that module. For example: `import re` lets you import the regular expressions module. You can rename a module as you import it: `import re as regex4life` if you want to show how much of a fan of regex you are, for example. Or just if you want to avoid a clash between names that have already been defined and the module name.

You can also import part of a module, for example: `from string import ascii_lowercase`. This allows you to use that part directly, without having to use `string.ascii_lowercase`. (It also saves you from importing the *entire* library.)

One thing that you will see, but should *never* do is `from re import *`. That asterisk tells Python to import everything from the `re` module into your namespace. But that also means that you might be overriding functions or variables that you've defined before! After all: it's unclear how much you're importing and what all these new things are called.

3.2 String: useful tools to work with strings

The `string` module doesn't add much to the string functionality in Python, but it does provide some useful constants (copied from the docs):

- **string.ascii_letters** The concatenation of the ascii_lowercase and ascii_uppercase constants described below. This value is not locale-dependent.
- **string.ascii_lowercase** The lowercase letters 'abcdefghijklmnopqrstuvwxyz'. This value is not locale-dependent and will not change.
- **string.ascii_uppercase** The uppercase letters 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. This value is not locale-dependent and will not change.
- **string.digits** The string '0123456789'.
- **string.hexdigits** The string '0123456789abcdefABCDEF'.
- **string.octdigits** The string '01234567'.
- **string.punctuation** String of ASCII characters which are considered punctuation characters in the C locale.
- **string.printable** String of ASCII characters which are considered printable. This is a combination of digits, ascii_letters, punctuation, and whitespace.
- **string.whitespace** A string containing all ASCII characters that are considered whitespace. This includes the characters space, tab, linefeed, return, formfeed, and vertical tab.

Use these constants whenever you want to loop over all characters or all numbers. As a general rule: **don't implement stuff that already exists in Python.** At the very least it will save you from embarrassing typos (e.g. missing letters in the alphabet).

3.2.1 Re: regular expressions

Regular expressions provide a very powerful way to search for patterns in text. The Python documentation already offers a great how-to that tells you all you should know about regular expressions. We will practice regular expressions in the notebooks.

3.2.2 Datetime: to work with dates

Read the documentation.

3.2.3 Collections: useful classes to store data

The collections module provides specialized classes that build on the core classes in the standard library. You can use them by importing them from the `collections` module, e.g. `from collections import namedtuple`. The most useful ones are listed below.

- **namedtuple** is an extension of the `tuple` class. It enables you to make your code more explicit and self-documenting, by making custom tuples. Suppose we were working with coordinates a lot. Then you could use `namedtuple` to create a `Point` object like this: Point = namedtuple('Point',['x', 'y']). Now every time you want to work with coordinates, you can use `Point` with the relevant coordinate values to instantiate a new point. E.g. `Point(5,8)` returns a `Point` object with attributes `x` and `y`, where the value of `x` is 5 and the value of `y` is 8.
- **defaultdict** is an extension of `dict` that lets you create dictionaries with default values. E.g. `d = defaultdict(list)` creates a dictionary where the standard value is an empty list. So you could immediately add values to that list for any key, without checking whether the key is in the dictionary already.
- **Counter** is an extension of `dict` that makes counting much easier. You can initialize it with an iterable to immediately count all the objects in the iterable, e.g. `c = Counter([1,2,3,4,5,6,5,4,35,3,2,2,4,6,7,8])`, or you can initialize an empty Counter and update the counts. E.g. `c = Counter()` followed by `c.update([1,2,3,4,1])`.

3.2.4 Math: everything you need for basic math

Read the documentation. If the math module isn't enough, then numpy or scipy might be something for you. (E.g. if you want to use matrices, or compute correlations or other statistical measures.)

3.2.5 Random: shuffle lists, sample data, or generate random numbers

The `random` module provides all kinds of randomizations. After you've imported `random`, you can:

- **Shuffle a list** in place using `random.shuffle(the_list)`
- **Sample a subset** using `random.sample(the_list, 25)`. You can also use this function as a shuffling function. If `n = len(the_list)`, use `random.sample(the_list, n)`.
- **Generate random numbers** either use `random.random()` to generate a number between 0 and 1, or use `random.choice(range(30))`.
- **Seed the random number generator** to make your results reproducible. Just use `random.seed(12345)` right after importing the `random` module and it will use that seed number to generate pseudorandom numbers that will be the same each run.

3.2.6 Itertools: loop over your data

The `itertools` module has too many useful functions to list here. Read the docs

3.2.7 Functools: advanced function manipulation

The `functools` module has useful meta-functions that you can use to manipulate existing functions. These are the two most common ones for me.

- `@functools.lru_cache()` is a decorator that memorizes input-output combinations for a function. This can speed up your code tremendously if you frequently call a particular function with the same input.
- The `partial` function is useful to 'fill in' an argument in a function, so that you won't have to type as much.

```
from functools import partial

def hello(source, target):
    "Tell someone hello."
    print('Hello,', source, '\nGreetings,', target)

hello_from_emiel = partial(hello, target='Emiel')

# This will print:
# "Hello Marten
# Greetings, Emiel"
hello_from_emiel('Marten')
```

3.3 os.path: manipulate paths

Windows machines have paths that look like `C:\\some\folder`, whereas UNIX-based operating systems have paths that look like `/some/folder`. `os.path` is a very useful library if you want to create OS-independent paths, so that your scripts will work on any computer without modifications. Here's how to use it:

```
import os
# Suppose that you have a set of texts in ./data/corpora/exciting corpus.
# If you want to make an OS-independent path to that folder, use:
relative_path = os.path.join('data', 'corpora','exciting_corpus')
absolute_path = os.path.abspath(relative_path)
```

This module is very useful in combination with `glob`.

3.4 Glob: find files

The `glob` module is useful to list all the files in a particular directory. After importing the module with `import glob`, you can find files using the `glob.glob()` function. For example, if you have a folder called 'data' with different kinds of files, and you want a list of all the text files, you might use `glob.glob('data/*.txt')`. The asterisk serves as a wildcard that matches any filename. You can also use the asterisk to match folders, e.g. `glob.glob('*/*.txt')`.

3.5 Csv: manipulate csv and tsv files

The `csv` module is very useful if you have spreadsheet-like data. Basic usage is as follows:

```
import csv

header = ['Name','Age']
rows = [['John', 5], ['Mary', 6], ['Bob', 12], ['Alice', 13]]

with open('Ages.tsv','w') as f:
    # If you don't specify a delimiter, the file will be a standard CSV file.
    # But now we specified a tab delimiter, the file is a TSV file instead.
    writer = csv.writer(f, delimiter='\t')
    writer.writerow(header) # for a single row
    writer.writerows(rows) # for multiple rows
```

The module also contains a `DictReader` and a `DictWriter` object that allows you to read and write rows using dictionaries. Very useful! Read the docs here.

3.6 JSON: manipulate json files

JSON is a very common file format on the internet, and it's useful to store data (lists, dicts, numbers and strings – no sets or tuples!) in a readable format.

Here's how to load a JSON file:

```
import json

with open('filename.json') as f:
    obj = json.load(f)
```

Writing out JSON files is equally simple:

```
import json

obj = ['bla', 2.0, {1: 'apple', 2: 'pear'}]

with open('filename.json','w') as f:
    json.dump(obj, f)
```

If you ever need to pretty-print a JSON object: `json.dumps(obj, indent=4, sort_keys=True)` returns a string. You can turn that string back into a JSON-serializable object by using `json.loads(str_representation)`.

3.7 Pickle: save python objects

The `pickle` module lets you save Python objects to your computer, so that you can reload them later. This may be useful if those objects take a long time to create. A downside of pickling objects, as opposed to saving your data in CSV/TSV/JSON/TX-T/XML format is that `.pickle` files are not human-readable. Moreover, you should never try to load a `.pickle` file from an untrusted source, as it may contain malicious code that will be executed when you load the file. Finally, pickle files are different for each version of Python, so they are not the best format for sharing data.

```
import pickle

# Create a dictionary
d = {'a':0, 'b':1, 'c':3}

# Write it to a pickle file:
with open('dictionary.pickle','w') as f:
    pickle.dump(d,f)

# Read a pickle file and load the object:
with open('dictionary.pickle') as f:
    obj = pickle.load(f)
```

3.8 Gzip: compressing files

If you're working with a lot of data, your files can become really big. To save space on your hard drive, or to be able to share your results without having to upload huge files, you can use the Gzip module. Gzip is a compression format that is well-suited to make text, xml, and csv data really small. Here's how it works:

```
import gzip

# If you use the 'w' or 'wb' mode then you need to encode the string:

with gzip.open('test.txt.gz','wb') as f:
    f.write('Hello, world!'.encode('utf-8'))

# But you don't have to if you open the file in text mode.
# See: http://stackoverflow.com/a/27206278/2899924

with gzip.open('test.txt.gz','wt') as f:
    f.write('Hello, world!')

# And you can even use the CSV module if you open the file in text mode:
import csv

rows = [list(range(10))] * 10
with gzip.open('test.csv.gz', 'wt') as f:
    writer = csv.writer(f)
    writer.writerows(rows)
```

Reading from gzip files works in a similar way. Read the docs for more.

3.9 Doctest: Enhance your docstrings and test your functions

Doctest is a module that enables you to automatically check whether your functions produce the output that you expect. It works like this:

```
# Example in /Examples/doctest-example.py

def hello(name):
    """
    Function that returns a greeting for whatever name you enter.

    Usage:
    >>> hello('Emiel')
    'Hello, Emiel!'
    """
    return ''.join(["Hello, ", name, '!'])
```

The docstring shows how to use the function, and the expected output of the function. The Doctest module allows you to test whether you actually get that expected output when you use the function as in the example. You can try this yourself. If you create a Python file with this function and run doctest (`python -m doctest -v doctest-example.py`), it should print the following message:

```
Trying:
    hello('Emiel')
Expecting:
    'Hello, Emiel!'
ok
1 items had no tests:
    doctest-example
1 items passed all tests:
   1 tests in doctest-example.hello
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Of course this is a trivial example, but it's very useful if you're working on a larger codebase. Especially if you want to add new functionality without changing the behavior of your functions. Then you really want to be able to check that everything still works as expected.

3.10 Standalone scripts and the Argparse module

While we've mostly been working with notebooks so far, you can also just write python scripts as standalone files. Basically it's just a plain text file with python code (called a 'script'), saved with the `.py` extension. Here's a very minimal example:

```python
def main():
    "The main function."
    print("Hello, world!")

if __name__ == "__main__":
    main()
```

If you save this file as `helloworld.py`, you can either import it (`import helloworld`) or run it from the command line by typing `python helloworld.py`. The if-statement is there so that it will only run the main function if the script is run from the command line. If you import the file, it won't print `"Hello, world!"` unless you use `helloworld.main()`.

If your program becomes bigger, you will probably want to pass arguments to the script from the command line. (Maybe you want to say "Hello" to a specific person!) The best thing to do then is to look into the `argparse` module. Here's how you set it up:

```python
# import the module
import argparse

# Initialize the parser
parser = argparse.ArgumentParser()

# Add arguments to be specified on the command line:
parser.add_argument("--target",
                    help="Person you want to say ’hello’ to.",
                    type=str)
parser.add_argument("--source",
                    help="Who is saying hello?",
                    type=str)

def hello(source, target):
    """
    The main function that says ’hello’ to the target,
    with greetings by the source."""
    print("Hello", target)
    print("Greetings from", source)

if __name__ == "__main__":
    args = parser.parse_args()
    hello(args.source, args.target)
```

You can test whether your code works by providing it with some arguments like this:

```python
import argparse

parser = argparse.ArgumentParser()

parser.add_argument("--target",
                    help="Person you want to say ’hello’ to.",
                    type=str)
```

```
parser.add_argument("--source",
                    help="Who is saying hello?",
                    type=str)

arguments = '--source Emiel --target Marten'.split()
print(parser.parse_args(arguments))
# This will print Namespace(source='Emiel', target='Marten')
```

For full reproducibility, it's recommended that you store all the arguments in a log file so that you'll always know exactly how the program was called. Here is a longer tutorial on the use of argparse.

Chapter 4

Standard recipes

Write!

Chapter 5

Remaining issues

This chapter is the last one covering Python-internals. After this, you know most of what there is to know about the language. The topics in this chapter may be considered *advanced*, but understanding them is very rewarding.

5.1 Python names and values

It's important to understand the way that Python assigns values to variables. Consider the following piece of code:

```
x = [1]
y = x
y.pop()
```

What is the value of x? The answer may surprise you. Click here and copy-paste the code to see what happens when you run this code.

Ned Batchelder has a great video about names and values that discusses the reason for this behavior.

5.2 Comprehensions

It's possible in Python to define lists using a for-loop. It works like this:

```
words = ['monty','python','spam','spam','spam']
my_list = [word.upper() for word in words if word.startswith('sp')]
print(my_list)
```

This prints: `['SPAM', 'SPAM', 'SPAM']`

As you loop over a sequence (in this case a list of words), you can check if the items fulfill some sort of condition (in this case: starting with the letters SP). If so: convert them to upper case and add them to the list. The result is a list with all the words matching that condition. This way of generating a list is called a *list comprehension*.

You can do the same thing with sets; just replace the rectangular brackets with curly brackets (and here I left out the conversion to upper case):

```
words = ['monty','python','spam','spam','spam']
my_set = {word for word in words if word.startswith('sp')}
print(my_set)
```

This prints `'spam'`, because lists cannot contain duplicates.

You can also create dictionaries using comprehensions. Here is an example:

```
# Store the result of multiplying a number by two in a dictionary:
doubles = {i: i*2 for i in range(20)}
```

But often the value isn't the output of a function or some other operation. Rather, you have keys and values that you want to store in a dict somehow. This is one way to do it:

```
last = ['Chapman', 'Cleese', 'Gilliam', 'Idle', 'Jones', 'Palin']
first = ['Graham', 'Cleese', 'Terry', 'Eric', 'Terry', 'Michael']

names = {last:first for last,first in zip(last,first)}
```

Of course in this specific case you could also do `dict(zip(last,first))`, but it is useful if there are particular conditions that you want to specify. For example, all the members who are called Terry:

```
last = ['Chapman', 'Cleese', 'Gilliam', 'Idle', 'Jones', 'Palin']
first = ['Graham', 'Cleese', 'Terry', 'Eric', 'Terry', 'Michael']

names = {last:first for last,first in zip(last,first) if first == 'Terry'}
```

When should you use a list comprehension?

- When you want a list/set/dict that can be defined in terms of some other iterable (or multiple iterables)

As we've seen above, there are two general modifications that you can make in a comprehension:

- Restrictive: use an if-statement to select particular items.
- Transformative: use a function or operation to modify the items in your newly defined list/set/dict.

Finally, you can use *nested for-loops* in comprehensions:

```
# Pairs of numbers between 0 and 10 that are not equal to each other.
# I like to align the for-loops so that it's clear to the reader how deep the
# nesting goes.
unequal_pairs = [(i,j) for i in range(10)
                       for j in range(10) if not i == j]

# This is equivalent to:
from itertools import product
unequal_pairs = [(i,j) for i,j in product(range(10),range(10)) if not i == j]

# But of course you could keep nesting:
unequal_pairs = [(i,j,k) for i in range(10)
                         for j in range(10)
                         for k in range(10) if not i==j==k]
```

5.3 Generators

Generators are like a combination of functions and sequences: when you call them, they return an iterable that you can loop over only once. The huge advantage of generators is that you never have all items in the iterable in memory at once. Here is a re-implementation of `range()` to give you an idea:

```
def new_range(n):
    """
    Limited re-implementation of range(). Yields all numbers up to n.
    """
    x = 0
    while x < n:
        yield x
        x += 1

for i in new_range(10):
    print(i)
```

This will print all the numbers from 0 up to (but not including) 10. But notice that the numbers aren't stored anywhere! Once you increment x, the old value is gone forever. But sometimes that's just what you want to do: go over everything, and throw away the data once you've extracted the relevant information. Else your computer's memory might be full and can't handle the new data anymore.

Instead of functions, you can also define generators like this:

```
squared_numbers = (x*x for x in range(10))
```

Or even use generator arguments:

```
# Sum of all squared numbers below 10.
sum(x*x for x in range(10))

# Dictionary of numbers and their squares.
dict((x,x*x) for x in range(10))
```

Generators provide *the* way to save memory. So if you're using some collection of items only once, or if you can't keep everything in memory, generators are the way to go. But note that there is a trade-off between time and memory: whatever you don't have to load into memory every time you need it saves you time. So think about how you choose to load your data.

If you haven't already, now is the time to watch Ned Batchelder's *loop like a native*.

5.4 Classes

As you've already seen, Python is an object-oriented language. It has basic classes, like lists, strings, dictionaries, and sets that all have their own methods. And it's fairly easy to extend those classes to provide new functionality. Or to create completely new classes (inheriting from `object`).

The `Counter` and `defaultdict` classes are examples of extensions of the dict class, and the `namedtuple` class is an extension of `tuple`. The Python tutorial has a good section on classes.

We will not spend a lot of time on defining classes. You should know that they are there, but for our purposes we will generally not be needing to define new ones. Watch Raymond Hettinger's talk on classes, though. There are some really good ideas in it.

5.5 External modules

While working on a problem, it's always worthwile to look around online to see if someone else has worked on something similar (or even on the same problem). Often you will find that, yes, someone has worked on something similar before, and there's even a module out there that is super useful and would save you a lot of work. Awesome! Now how to go from here?

The first thing to do is see if you already have the module installed. Just try to import it and see what happens. Worst case, you'll see something like this:

```
>>> import unknown_module
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'unknown_module'
```

If the module is not already installed, and you're using anaconda, use `conda install MODULENAME`. (Where MODULENAME is replaced by whatever the module is called.) If that doesn't work, try `pip install MODULENAME`. This usually works. If it doesn't, look for instructions on how to install the module.

Chapter 6

Chapter 5 – On your own

Now that you know the basics of Python, let's talk strategy. How do you tackle new problems? What do you do when you get stuck? What if your program takes ages to finish? That's what this chapter is about: tips and tricks that we've found helpful over the years.

6.1 How to tackle any problem

- Start small, with one data point.
- Start from the problem, not from the code. Don't think "I need to loop over this and then . . . ", but think "I want to take X, and get Y. How do I get from X to Y?"
- Start with what you know. E.g. if you want to do something with a string, make a variable with a suitable name and check whether any of the string methods might be useful.
- Divide the problem into steps: what do you need to do first? Solve that thing first, and only once you've verified that it works move on to the next step.
- Use print-statements extensively as you are moving forward. Make sure that you know what your code does at every point in the script.
- Document your code as you write. This forces you to explain what you are doing.
- Be conscious of your assumptions, and spell them out. (You could even write test cases with these assumptions in mind.)
- Keep to the style guide, and write clear function names. This forces you to think about what it is you're doing. Variables called `a, b, c` obfuscate problems. (Unless these characters have a specific meaning in the context.)
- If you get stuck, try to explain to yourself what every part of your code does. This also works with others: see pair programming and rubber duck debugging.

6.2 How to deal with large amounts of data

When you have to deal with large amounts of data, it's important to think about scalability: what happens when you run your script on thousands of files? There's a number of constraining factors (**bottlenecks**) that influence how long it will take for your program to run (or even whether it can complete). Here they are:

- **CPU speed** If your processor runs at 100% of its capacity, it is likely that processing speed is the limiting factor. There are two things you can do:

   1. Optimize your code (see below),
   2. Parallelize your code (look into the multiprocessing package).

- **Memory size** Sometimes a program can take up all of your computer's working memory (RAM). Some computers are configured to use part of your hard drive as additional memory ('swap memory'), but this has the downside of being much slower. If your computer runs out of memory, you will get a `MemoryError`. There are three things you can do:

   1. Optimize your code (use generators and other techniques that only keep relevant data in memory and discard everything when it's no longer needed).
   2. Preprocess the data first so that your program only works on the relevant data (this can be combined with 1).
   3. Write the data you don't need all the time to the hard drive.

  (On the flip side, if you have plenty of memory, you can sometimes exchange memory for speed: keep stuff in memory so that everything is quickly accessible. Do note that this strategy doesn't scale well.)
- **I/O (input/output)** When your program does a lot of reading and writing, you may see that the CPU runs at less than its full capacity. If this is the case, the capacity of your hard drive might be the limiting factor. One solution is to buffer the data: keep all the data-to-be-written in a list (or some other container) and write everything out at once (or every time a threshold amount is met).

6.3 Optimizing your code

- Read your code again, preferably out loud. Try to simulate in your head what happens with some toy input data. Are there any things that stick out? Fix these things first.
   - Which parts are tedious to explain?
   - Are there any repetitions?
   - Are there any small helper functions that can be integrated back into the main function because they are oddly specific?
- Check that you're loading prerequisite files only once, and not each time your script processes a new document.
- Python built-ins are usually much faster than anything you can write by yourself. Check if you're making use of them.
- If you have many list membership checks in your code, check whether you could replace those lists by sets.
- If you only loop over a sequence of items once, try to use a generator instead.
- Are you calling a function many times with the same arguments? Then the `lru_cache` function might be useful (see here).
- If you run out of memory in places where variable reference to one large object is replaced by a reference to another large object, the `del` statement might be useful (so that Python never keeps *both* objects in memory).
- If you use a lot of if-statements to check for corner cases, consider using `try... except` instead. This gets rid of all the unnecessary checks.
- Speaking of unnecessary checks: look at the order of your `if... elif...` statements. If you put the most common case first, and the least common case last,

the interpreter spends less time checking all the statements (because once it finds a match, it can skip the rest).

- Try to think about cheap (processing-wise) heuristics that indicate whether or not a sentence/file/??? is suitable for further (more intensive and time-consuming) processing.

Chapter 7

Frequently asked questions

7.1 Which brackets should I use?

- **Parentheses** are used to define *tuples* and to call functions.
- **Curly braces** are used to define sets and dictionaries.
- **Square brackets** are used to define *lists*, for indexing, slicing, and to get the value for a particular key in a dictionary.

Kind	Example	What does this do?
Tuple	`x = (1,2,3)`	Make `x` refer to a *tuple* with the values 1,2, and 3.
Function	`print("Hello, world!")`	*Call* the print function to display the string "Hello, world!"
Dictionary	`d = {'apples': 1, pears: 2}`	Make `d` refer to a dictionary containing the keys "apples" and "pears", and their corresponding values.
Set	`x = {1, 2, 3}`	Make `x` refer to a *set* containing the values 1,2, and 3.
List	`x = [1,2,3]`	Make `x` refer to a *list* with the values 1,2, and 3.
Indexing	`y = x[0]`	Make `y` refer to the first element of `x`
Slicing	`y = x[:2]`	Make `y` refer to the first two elements of `x`
Dictionary	`y = d['apples']`	Get the *value* corresponding to the *key* "apples" from the dictionary `d`.

7.2 What is the difference between functions and methods?

- **Methods** are functions that are defined for specific types of objects. One example is the method used to make a string lowercase: `'Word'.lower()`. Methods are always called using the dot-notation.
- **Function** is a more general term. Functions are like variables, but instead of referring to specific values, they refer to chunks of code. If you call a function (using the parentheses), the corresponding chunk of code will be executed.

7.3 What is the difference between Jupyter and Python?

Python is a programming language that you can use to give instructions to the computer. If you run a bit of Python code, your computer will use a *Python-interpreter* to understand what you would like the computer to do. The official Python-interpreter is called *CPython*.

Jupyter is a program that allows you to run Python code through a browser-based interface. Behind the scenes, it calls CPython to run your code, and then it returns the result. Jupyter works using a special file format called 'Jupyter notebooks' (previously known as 'iPython notebooks', which is why the file extension is still `.ipynb`). We really like this programming environment, because it is very powerful, and you can use it to write reports about your research. Because these notebooks are executable, you can give them to someone else, and they'll be able to reproduce your results. This is extremely useful for scientific research.

Anaconda provides a Python distribution with loads of useful pre-installed modules. The developers of Anaconda have tried to make it as easy as possible to use Python on any major operating system. Before Anaconda, it was very hard to install some modules if you happened to have the wrong operating system, or the wrong computer configurations. With Anaconda, we haven't had any issues with students not being able to install Python on their laptops.