# Python introduction

Georg Huhs  
Barcelona Supercomputing Center

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Basic description</strong></td>
<td>2</td>
</tr>
<tr>
<td>1.1 What kind of programming language is Python?</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Paradigm</td>
<td>2</td>
</tr>
<tr>
<td>1.3 A matter of interpretation</td>
<td>2</td>
</tr>
<tr>
<td>1.4 What about performance</td>
<td>3</td>
</tr>
<tr>
<td>1.5 Python 2 or 3?</td>
<td>4</td>
</tr>
<tr>
<td>1.6 So, how to get started?</td>
<td>4</td>
</tr>
<tr>
<td><strong>2 Main features</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 Syntax</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Types &amp; variables</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Functions</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Classes</td>
<td>8</td>
</tr>
<tr>
<td>2.5 Modules</td>
<td>9</td>
</tr>
<tr>
<td><strong>3 The Python ecosystem</strong></td>
<td>11</td>
</tr>
<tr>
<td>3.1 The Python core</td>
<td>11</td>
</tr>
<tr>
<td>3.2 External packages</td>
<td>11</td>
</tr>
<tr>
<td>3.3 Virtual environments</td>
<td>12</td>
</tr>
<tr>
<td><strong>4 How to take advantage of Python</strong></td>
<td>13</td>
</tr>
<tr>
<td>4.1 Zen of Python</td>
<td>13</td>
</tr>
<tr>
<td>4.2 The famous PEP8</td>
<td>14</td>
</tr>
<tr>
<td>4.3 Writing idomatic Python</td>
<td>15</td>
</tr>
<tr>
<td><strong>5 Code quality matters</strong></td>
<td>19</td>
</tr>
<tr>
<td>5.1 Documentation</td>
<td>19</td>
</tr>
<tr>
<td>5.2 Testing</td>
<td>20</td>
</tr>
</tbody>
</table>
1 Basic description

1.1 What kind of programming language is Python?

There are many associations, but also questions related to Python and its usage:

- scripting language / console
- (just?) good as glue code
- high/low performance language
- good for prototyping (only?)
- object oriented
- interpreted (slow?)

For clarifying one can start searching at the very base, the Python.org homepage. There, the FAQ answers the question “What is Python good for?” with:

*Python is a high-level general-purpose programming language that can be applied to many different classes of problems.*

Its objective is to make high level programming fun and efficient. Of course these two are interlinked in a positive cycle - reaching your goals efficiently is fun and satisfying, and having fun helps with being efficient.

The means to achieve this are

- clear and natural-language-like syntax
- large eco-system, including many libraries
- easy integration of other code
- being general (purpose)

1.2 Paradigm

Python is a multi-paradigm language. Mostly used for imperative/procedural (Fortran-, C-style) or object-oriented (C++, Java) implementations, but it also allows functional and reflective code, and even more by language extensions.

1.3 A matter of interpretation

Fist one has to notice that a language definition, like Python as such, does not define what a system does with code written in it. This is part of the language implementation.

In the most used Python implementation (CPython) the code is compiled into an intermediate representation each time a change in the source is detected before execution. This intermediate representation is bytecode, much more efficient to process than

[https://docs.python.org/3/faq/general.html](https://docs.python.org/3/faq/general.html)
the original code, but still interpreted by a virtual machine, in contrast to classically compiled code, which is executed directly on hardware. A way of describing this is byte-code interpreted. Other options for processing, for example just-in-time (JIT) compilation, are available.

1.4 What about performance

Maybe the most relevant example in our field regarding performance of Python is GPAW, a massively parallel electronic structure code mainly written in Python. The “mainly” comes from a relatively small rest of core-routines written in C, motivated by, as one developer expressed it:

Python is good at everything, except multiplying numbers quickly.

Of course one of the main tasks of our scientific codes is multiplying numbers, but does this disqualify Python for this purpose? The key to this is, that the amount of computations is not reflected in the amount of code-lines. Actually, experience shows, that it is rather easy to separate some small number-crunching functionality, and have all the control structures around it in a nice code. This is exactly what has been done with high-performance numerical libraries like ScaLAPACK, and codes like GPAW extend this principle. Figure 1 shows, that only about 10% of their code consist of number-crunching routines written in C.
1.5 Python 2 or 3?

There are two main Python versions around. Obviously version 3 is newer, but version 2 is still very present and at many systems even the default. So what to choose?

As the Python.org wiki puts it\footnote{https://wiki.python.org/moin/Python2orPython3}:

*Python 2.x is legacy, Python 3.x is the present and future of the language*

The latest Python 2 release is 2.7 from 2010 and as one can guess, the more developed language is Python 3, with its current release 3.6 (2016). Using the old version it has no advantages, but might be necessary if strictly needed libraries are only available for Python 2.

In this introduction only the Python 3 standard is used.

1.6 So, how to get started?

Learning the syntax, as first step, takes only little effort. After that one should really invest in learning how to do things the pythonic (see later) way and ask as often, very often “shouldn’t there be an easier way?”

A lot of information for getting started is provided by the official python.org web page:

- **Central Python documentation**  Contains complete references as well as introductory material (tutorial...).
  
  [https://docs.python.org](https://docs.python.org)

- **Wiki**  A lot of information “around” Python, like beginner’s guides, a list of beginner’s errors, recommendations for books and tools, ...
  
  [https://wiki.python.org](https://wiki.python.org)

- **FAQ**  Short answers to the basic questions treated here (and many more)
  
  [https://docs.python.org/3/faq/general.html](https://docs.python.org/3/faq/general.html)

Python has a very rich community, leaving a lot of advice in the form of blogs and discussions on the internet. And for basically any beginner’s-question one can assume, that it has been stated already somewhere.
2 Main features

2.1 Syntax

Python prefers a visual and textual way instead of using special characters.

- Scope of routines, loops, ... is defined by indentation

```python
sum = 0
i = 1
while i <= 10:
    sum = sum + i
    i += 1
response = "The sum of all integers from 1 to 10 " \n    "might be " + str(sum)
print(response)
```

Line 8 is a continuation of the previous line. The indentation there is not strictly necessary, but recommended for better readability.

- Natural language:
  The code above is actually bad style (more about that later). An alternative is the following:

```python
sum = 0
for i in range(1, 11):
    sum = sum + i
```

Besides being more compact, it expresses what happens in a more natural way.

- Special constructs for generators:

```python
the_sum = sum(i for i in range(1, 11))
```

This is still doing the same, but written in the most Python-like way, using a particular syntax for generating a series of values.
2.2 Types & variables

As already seen, Python does not use explicit typing. There are no primitive types, which means that everything that can be assigned to a variable is an object. For example, adding two numbers can, besides using the + operator, be done by:

```
(1) ...add...(3)
```

Some classes, for example those that are basic types in other languages (float, int, and complex, ...) are immutable.

In Python containers are extremely helpful and extensively used. Such types are for example:

- **list**

```
my_list = [1, 2, 3]
the_second = my_list[1]
the_last = my_list[-1]
all_but_the_first = my_list[1:]
```

- **dictionary**

```
language_description = {"name": "python",
"version": 3}
language_description["useful"] = True
if language_description["useful"]:    
print("seems useful: " + language_description["name"])
```

- **tuple**

```
(b,a) = (a,b)
```

2.3 Functions

An example for how to define a simple function (the output is given as comment below each print function):

```
def times_two(input):
    output = input * 2
    return output

a = 3
double_a = times_two(a)
print(double_a)
# 6

b = complex(1,1)
double_b = times_two(b)
print(double_b)
# (2+2j)
```
Functions are also objects, which implies:
- a variable can be assigned to a function ("renaming")
- functions can be defined within functions.
- functions can be a parameters or return values of other functions.

Python provides a compact representation of functions working on functions (equal to functionals in mathematics), the *decorators*.

A recurring question is if Python works with *call-by-value* or *call-by-reference*. The discussions about the answer are often hilarious, which shows that probably the best idea is to just understand its behavior without classifying it with terms which are often enough not even clearly defined.

Keeping in mind that everything is an object (= an instance of a class), variables are just names attached to an object. An assignment is the process of binding this name to the object. Where this name is valid, is defined by the scope of the assignment.

One way of seeing this is putting lables, (like price tags) to objects. In this sense re-assigning a variable equals moving the label to another object. Applying this to function calls we get:

```python
def change(input):
    input += 1
a = 4
change(a)
print(a)  # 4

def change(input):
    input[0] += 1
a = [4]
change(a)
print(a)  # [5]
```

In both cases, when entering the function `change`, the names `input` and `a` are labels for the same object.

In the left case the object is an integer and thus immutable, so the `change` function assigns the name `input` to a new object, which leaves the “outside” `a` unchanged.

In the second case, the object is a list, a mutable object. It allows to change its content, which is exactly what is done in the function. Since `a` is related to the same object, it also shows the change *inside* the object.

The good news about this is, that when writing code in a clean style, hardly any questions or surprises show up. If you constantly run into doubts of the behavior of your code, most probably not the language, but its usage is the problem.

---

3A good explanation with some vintage ASCII art can be found at [http://python.net/crew/mwh/hacks/objectthink.html](http://python.net/crew/mwh/hacks/objectthink.html)

4Nicely depicted at [http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html#other-languages-have-variables](http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html#other-languages-have-variables) and the following chapter.
2.4 Classes

An example for class definition and usage:

```python
class Multiplier(object):
    name = "multiplier"
    def __init__(self, factor):
        self.factor = factor
    def multiply(self, input):
        output = input * self.factor
        return output

times_two = Multiplier(2)
a = 3
double_a = times_two.multiply(a)
print(double_a)  # 6

b = complex(1,1)
double_b = Multiplier.multiply(times_two, b)
print(double_b)  # (2+2j)

print(Multiplier.name)  # multiplier
```

Line 1 defines the name of the class and its parent. Python allows multiple inheritance.

Line 3 defines a variable which is the same for all instances of the class. It is used in line 25.

Lines 5-6 are the constructor. Python has a set of predefined routines names of the form `bla` with special meanings. Those are overloaded for providing individual functionality. In this case we use it for specifying a multiplication factor to be set by the user.

Lines 8-10 define a class routine. With the exception of static routines, the first argument is always the class instance. Here we see that the variable defined in the constructor is available in the `multiply` routine.

Line 13 creates an instance of the `Multiplier` object.

Line 16 makes use of this object.

Line 21 does the same as line 16, but explicitly using the instance parameter.

There is no mechanism for making routines or variables invisible outside of the class. However, there is the convention that member names starting with an underscore (`_`) are meant to be “private”. This is based on the assumption, that programmers are mature enough not to need restrictions given by the language.
2.5 Modules

A module is a file containing Python definitions and statements. The multiplier class defined above can be such a module:

```
class Multiplier(object):
    name = "multiplier"
    def __init__(self, factor):
        self.factor = factor
    def multiply(self, input):
        output = input * self.factor
        return output
```

```
times_two = Multiplier(2)
times_three = Multiplier(3)
```

It contains the definition of a class and two instances. The keyword for accessing a module from any code is `import`, with the module name given by its filename:

```
import multiplier_module
from multiplier_module import Multiplier
from multiplier_module import times_three

a = 3
double_a = multiplier_module.times_two.multiply(a)
triple_a = times_three.multiply(a)
i_want_four = Multiplier(4)
four_a = i_want_four.multiply(a)
```

This code shows two ways of importing module functionality.

- **Line 1** makes the whole module visible. Then its content is accessed via the module name, as shown in line 6.

- **Lines 2 and 3** import specific members of the module, which makes them directly accessible, demonstrated in lines 7 and 9.

It is also possible to import everything by using the asterisk (`*`) character, however, this is regarded bad style because it does not explicitly show what functionalities are available, and can yield problems with namespaces.

---

9
Modules can also provide functionality for being **executed** from the shell:

```python
#!/usr/bin/env python3
class Multiplier(object):
    name = "multiplier"

def __init__(self, factor):
    self.factor = factor

def multiply(self, input):
    output = input * self.factor
    return output

times_two = Multiplier(2)

if __name__ == "__main__":
    import sys
    input = int(sys.argv[1])
    the_double = times_two.multiply(input)
    print("Your input times 2 equals: \{0\}.format(the_double))
```

This can be called from the shell in two ways:

```
$> python3 multiplier_executable.py 3
$> ./multiplier_executable.py 3
```

**Line 1** defines the interpreter and makes the second version of the call possible.

**Line 16** is where the magic happens. A module can access its own name in the variable \_\_name\_\_. If it is called as main program, its name is \_\_main\_\_. Thus the if-clause means that the code in its scope is only executed when calling `multiplier_executable.py` as executable. So this file can run as script, as well as serve as module to be included in other code.

**Line 17** includes `sys` from Python’s standard library, which is then used for getting the command line arguments in line 19.
3 The Python ecosystem

3.1 The Python core

*The Python Language Reference* can be found at:
https://docs.python.org/3/reference/index.html

Python comes with an extensive set of functionalities, called *The Python Standard Library*:
https://docs.python.org/3/library/index.html

3.2 External packages

Python encourages the distribution of useful developments by providing
- a centralized code repository, currently containing over 100.000 packages
- and a proper package management system making installations extremely easy

Recommendations about the usage of packages are given by the Python Packaging Authority (PyPA):
https://packaging.python.org/guides/tool-recommendations/

One example for a field where extensively using Python has been has been going hand in hand with the development of many libraries, is data visualization, nicely explained in a youtube video\footnote{https://youtu.be/OC-YdBz8L1w}.

3.2.1 The Python Package Index

The mentioned central package repository is called Python Package Index, or short PyPI, to be found at:
https://pypi.python.org

Some examples for packages heavily used in science are:
- **NumPy** array processing for numbers, strings, records, and objects
- **SciPy** scientific library for Python, e.g. numerical integration and optimization
- **matplotlib** 2D plotting library
- **SQLAlchemy** SQL toolkit and object relational mapper

One can also find packages that make important tools available. For example **spglib** is originally a C-code for working with crystal symmetries. There is a Python-package with the same name, providing bindings to the C library. Similarly the package **mpi4py** provides Python bindings for MPI.
### 3.2.2 Package management

Python’s recommended tool for managing packages is **pip**. It works as command line tool, pretty similar to Linux package managers.

The main commands are

- **pip install <package_name>** downloads and installs packages registered at PyPi. Also a local package can be provided.
- **pip install --upgrade <package_name>** upgrades an installed package.
- **pip list** shows which packages are currently installed.

### 3.3 Virtual environments

A virtual environment is an independent, encapsulated Python installation. This allows having several working environments, using different packages or different versions of them, even different Python versions.

The usual three steps of working with virtual environments are:

1. Create virtual environment (only once)
   
   There are several virtual environment tools available. The recommended one for Python3 is **venv**. A new virtual environment named *venv_name* (we indicate that this is a parameter by using the brackets) is set up by
   
   ```bash
   $ python -m venv <venv_name>
   ```

   This will create a folder which is used as Python installation and working directory, but is completely independent from the code to work on.

2. Activate environment

   ```bash
   $ source <venv_name>/bin/activate
   ```

   This still leaves the user in the shell, indicating the currently activated environment by writing its name at the beginning of the prompt.

3. Exit environment

   ```bash
   (<venv_name>) $ deactivate
   ```

A newly created virtual environment will contain only a few packages, which ones can be checked by using **pip list**. The whole flow looks like:

```bash
$ pyvenv <venv_name>
$ source <venv_name>/bin/activate
(<venv_name>) $ pip list
pip (8.1.1)
setuptools (20.10.1)
(<venv_name>) $ deactivate
```
4 How to take advantage of Python

So far basic principles for making it possible to use Python were presented. But there is no point in changing the tools one uses, if they don’t yield any improvement. In the case of Python, the language has the potential of making programming considerably more productive, but this potential has to be unleashed by the way of using Python.

This is well expressed in a blog entry called “Python is not Java”[6] about a code produced by people who had just switched from Java to Python:

So, the sad thing is that these poor folks worked much, much harder than they needed to, in order to produce much more code than they needed to write, that then performs much more slowly than the equivalent idiomatic Python would.

Making good usage of the language and doing things “the Python way” has even its own term: pythonic.

4.1 Zen of Python

Basic “instructions” for working in a pythonic way are given by the Zen of Python, which can be retrieved by typing in Python console:

```
import this
```

In full length this is:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren’t special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one– and preferably only one –obvious way to do it.
- Although that way may not be obvious at first unless you’re Dutch.

• Now is better than never.
• Although never is often better than *right* now.
• If the implementation is hard to explain, it’s a bad idea.
• If the implementation is easy to explain, it may be a good idea.
• Namespaces are one honking great idea – let’s do more of those!

4.2 The famous PEP8

Some important instructions for good Python-usage can be found in the Python En-
ehmenience Proposals, or short PEP[7]. Despite sounding pretty casual they define im-
portant processes, functionalities, and rules which are accepted as standard. Each of
those proposals has a fixed number, for example the zen of Python is PEP20.

Probably the most important PEP is PEP8, which is a reference for the Python cod-
ing style. In contrast to other languages this is a globally accepted standard, thus
also many tools around it have been developed, for example plugins for editors that
constantly check PEP8 compliance and inform about problems. There are even online
PEP8 checkers, for example [http://pep8online.com/](http://pep8online.com/)
4.3 Writing idomatic Python

In the following a few examples are given as demonstration. For going deeper in to this topic masses of resources are available online and in printed form. A nice one can be found in Jeff Knupps blog\(^8\), which contains also other topics in a short but yet solid form.

4.3.1 Lists & co

Some basic list and string processing:

```python
some_strings = ["This", "is", "not", "hello", "world"]

# we want a numerated list of the strings
index = 0
while index < len(some_strings):
    print("{0}: {1}".format(index, some_strings[index]))
    index += 1

# now the pythonic version
for (index, string) in enumerate(some_strings):
    print("{0}: {1}".format(index, string))

# to build a sentence
sentence = some_strings[0]
for string in some_strings[1:]:
    sentence += " " + string
print(sentence)

# now the pythonic version:
sentence = " ".join(some_strings)
print(sentence)
```

In general `enumerate` should always be used instead of manually managed list indices, as well as the `in` keyword for going through iterable objects. Consistently everything you would like to be iterable, is iterable.

There are many tools around for making list handling easy, for example `zip` for combining iterables.

\(^8\)https://jeffknupp.com/blog/2012/10/04/writing-idiomatic-python/
4.3.2 Generators and list comprehensions

Imagine we want the sum of all prime numbers up to the number given by \texttt{max}, given we have a function \texttt{isprime} that checks if an integer is a prime number. Then a generator expression can be used to formulate this in a compact way:

\begin{verbatim}
the_sum = sum(i for i in range(1, max) if is_prime(i))
\end{verbatim}

A generator expression evaluates the items one by one. If they are needed all at once, for example for keeping them in memory, a list comprehension can be used:

\begin{verbatim}
my_numbers = [i for i in range(1, max) if is_prime(i)]
the_sum = sum(my_numbers)
\end{verbatim}

4.3.3 EAFP and exception handling

EAFP stands for: "It’s easier to ask forgiveness than permission", in contrast to the LBYL “Look before you leap” concept.

Imagine the following birthday calendar:

\begin{verbatim}
bdays_basic.py
from dateutil import parser
from datetime import datetime
import math

birthdays = {
    "hans": "31.12.1966",
    "sepp": "02.04.1986",
    "lilli": "29.02.2001"
}

name = input("we are looking for: ")

date_bday = parser.parse(birthdays[name])
date_now = datetime.now()
age = math.floor((date_now - date_bday).days / 365.25)

print("{} is {} years old".format(name, age))
\end{verbatim}

We have a list of people with their birth dates as dictionary. Obviously the code provides a minimal command line interface reading a name from the user input, and calculating the current age of that person.

If the user asks for a person that is not present in the dictionary, the code fails with:

\begin{verbatim}
Traceback (most recent call last):
  File "/home/ghuhs/bse/presentations/20170801_python_intro/10.py", line 12, in <module>
    date_bday = parser.parse(birthdays[name])
KeyError: 'hh'
\end{verbatim}
The - coming from other languages - probably most intuitive way to deal with this is to check first if the name is in the dictionary, which is the LBYL way:

```
from dateutil import parser
from datetime import datetime
import math

birthdays = {'hans': '31.12.1966',
             'sepp': '02.04.1986',
             'lilli': '29.02.2001'}

name = input("we are looking for: ")

if name in birthdays:
    date_bday = parser.parse(birthdays[name])
    date_now = datetime.now()
    age = math.floor((date_now - date_bday).days / 365.25)
    print("{} is {} years old".format(name, age))
else:
    print("unknown name")
```

If now we ask for Lilli’s age, we notice that there is something wrong with the date:

```
File "/home/ghuhs/bsc/presentations/20170801_python_intro/10.py", line 12, in <module>
    date_bday = parser.parse(birthdays[name])
File "/usr/lib/python3.4/site-packages/dateutil/parser.py", line 1008, in parse
    return DEFAULTPARSER.parse(timestr, **kwargs)
File "/usr/lib/python3.4/site-packages/dateutil/parser.py", line 404, in parse
    ret = default.replace(**repl)
ValueError: day is out of range for month
```

So if we want to avoid also this case, we need to check if the given dates are valid. Moreover we should also ensure that the date string has a valid format, and so on. The resulting code would consist of more checking than actual functionality, distributed through the code, so that in the end the actual workflow is hard to see. Remember the zen: "Readability counts"!

Of course one could (and should) ask if Python does not provide all this checking. And of course it does, and its way of dealing with it is exactly the exception ValueError we see.
This leads us to the EAFP concept, which translates to: “write the standard behavior of your code and treat the exceptions.” Applying this we end up with:

```python
from dateutil import parser
from datetime import datetime
import math

birthdays = {
    "hans": "31.12.1966",
    "sepp": "02.04.1986",
    "lilli": "29.02.2001"
}

name = input("we are looking for: ")

try:
    date_bday = parser.parse(birthdays[name])
    date_now = datetime.now()
    age = math.floor((date_now - date_bday).days / 365.25)
    print("{} is {} years old".format(name, age))
except KeyError:
    print("name not found")
except ValueError as problem:
    print("invalid birthday date: " + str(problem))
```

This provides:

- readable code
- explicit treatment of errors (also in the zen)
- flexibility on the level of error handling, as one can also catch mistakes on a coarser level
- avoiding the need to access the dictionary twice, which furthermore avoids race conditions in situations where the object might change between the accesses
5 Code quality matters

Producing a software product which is presentable to (outside) customers/users as well as sustainable for (internal) developments is not a Python-specific requirement. Detailed descriptions would go beyond the scope of this introduction, so here only some entry-points for getting started are mentioned.

Key topics besides the design of the actual code are documentation and testing.

5.1 Documentation

Python provides the concept of docstrings. These are comments which are the first statement in module, function, class, and method definitions. Actually each of these entities should have a docstring, explaining what it does (not how).

A docstring can be any string, also going over multiple lines.

For example a docstring for generating a complex number could read:

```python
def complex(real=0.0, imag=0.0):
    """Form a complex number.
    
    Keyword arguments:
    
    - real -- the real part (default 0.0)
    - imag -- the imaginary part (default 0.0)
    """
```

The most established tool for autogenerating code documentation from these docstrings is Sphinx[^9], "the Doxygen of Python".

It has its own pretty extensive syntax based on reStructured Text (reST), allowing text formatting, images, tables, and so on. For example a Sphinx-compatible version of the documentation above is:

```python
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    
    - real: the real part (default 0.0)
    - imag: the imaginary part (default 0.0)
    - type real: any scalar number type
    - type imag: any scalar number type
    - return: an object representing a complex number
    - rtype: Complex
    """
```

[^9]: http://www.sphinx-doc.org
5.2 Testing

A JUnit-inspired unit testing framework is directly integrated into Python as part of the standard library.

The example given by the module documentation is:

```python
import unittest
class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        s = 'hello world
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

[https://docs.python.org/3/library/unittest.html](https://docs.python.org/3/library/unittest.html)