python-102 Documentation

Release 0.1

Ashwin Srinath

Feb 10, 2023

Contents:

This tutorial covers topics that are essential for scientific computing and data analysis in Python, but typically *not* covered in an introductory course or workshop.

These are the thing you *need* to know if you are writing software that meets any of the following criteria:

- You expect to be working on it for more than a couple of weeks.
- You expect that it will be composed of more than a hundred or so lines of code.
- You want it to produce results that can be trusted - for example, if you are publishing a research paper based on those results.
- You expect that it will be used by one or more other people.
- You are contributing to another project - e.g., an open-source software package.

CHAPTER 1

What you will learn

1. How to organize the code for your project, and how to make it an installable *package* rather than a loose collection of files.
2. How to write tests for your code so that you can be sure it always produces the correct answer, even as you make changes to it.
3. How to document your code so that it is easy for you and others to use and navigate.
4. How to improve the usability of your code.
5. How to improve the performance of your code.

CHAPTER 2

What you need to know

This tutorial assumes you know the very basics of programming with Python.

If you can write a loop and a function in Python, and if you know how to run a `.py` script, you should be able to follow this tutorial easily.

CHAPTER 3

What you need to have

If you plan to participate in the hands-on exercises, you will need:

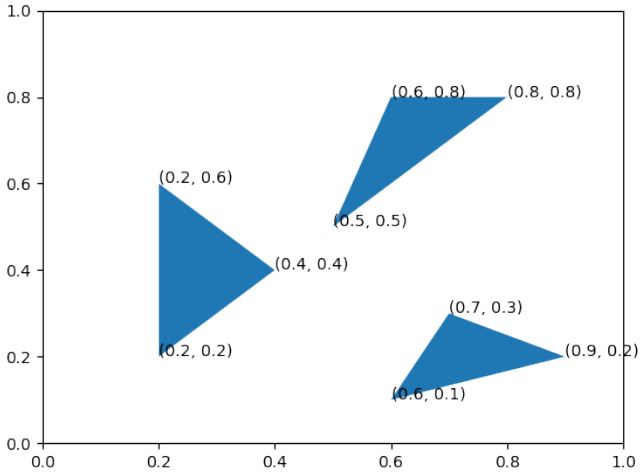
- A laptop with Anaconda installed on it
- 1 or more friends. It is **highly** encouraged to work in groups, so if you haven't already, please introduce yourself to your neighbour(s).

3.1 Organizing code for a Python project

A well structured project is easy to navigate and make changes and improvements to. It's also more likely to be used by other people – and that includes *you* a few weeks from now!

3.1.1 Organization basics

We want to write a Python program that draws triangles:



We use the the Polygon class of the matplotlib library and write a script called `draw_triangles.py` to do this:

Listing 1: draw_triangles.py

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.set_xlabel('x')
ax.set_ylabel('y')

patch = plt.Polygon([
    (0.2, 0.2),
    (0.2, 0.6),
    (0.4, 0.4)
])

ax.add_patch(patch)

ax.text(0.2, 0.4, '(0.2, 0.4)')
ax.text(0.2, 0.6, '(0.2, 0.6)')
ax.text(0.2, 0.4, '(0.2, 0.4)')

patch = plt.Polygon([
    (0.6, 0.8),
    (0.8, 0.8),
    (0.5, 0.5)
])

ax.add_patch(patch)

ax.text(0.6, 0.8, '(0.6, 0.8)')
ax.text(0.8, 0.8, '(0.8, 0.8)')
ax.text(0.5, 0.5, '(0.5, 0.5)')

patch = plt.Polygon([
    (0.6, 0.1),
    (0.7, 0.3),
```

(continues on next page)

(continued from previous page)

```
    (0.9, 0.2)
])

ax.add_patch(patch)

ax.text(0.6, 0.1, '(0.6, 0.1)')
ax.text(0.7, 0.3, '(0.7, 0.3)')
ax.text(0.9, 0.2, '(0.9, 0.2)')

plt.show()
```

Do you think this is a good way to organize the code? What do you think could be improved in the script `draw_triangles.py`?

Functions

Functions facilitate code reuse. Whenever you see yourself typing the same code twice in the same program or project, it is a clear indication that the code belongs in a function.

A good function:

- has a descriptive name. `draw_triangle` is a better name than `plot`.
- is small – no more than a couple of dozen lines – and does **one** thing. If a function is doing too much, then it should probably be broken into smaller functions.
- can be easily tested – more on this soon.
- is well documented – more on this later.

In the script `draw_triangles.py` above, it would be a good idea to define a function called `draw_triangle` that draws a single triangle, and re-use this function every time we need to draw a triangle:

Listing 2: draw_triangles.py

```
import matplotlib.pyplot as plt

def draw_triangle(points, ax=None):
    if ax is None:
        ax = plt.gca()
    else:
        fig, ax = plt.subplots()
        ax.set_xlabel('x')
        ax.set_ylabel('y')

    patch = plt.Polygon(points)
    ax.add_patch(patch)

    for pt in points:
        x, y = pt
        ax.text(x, y, '({}, {})'.format(x, y))

draw_triangle([
    (0.2, 0.2),
    (0.2, 0.6),
    (0.4, 0.4)
])
```

(continues on next page)

(continued from previous page)

```
draw_triangle([
    (0.6, 0.8),
    (0.8, 0.8),
    (0.5, 0.5)
])

draw_triangle([
    (0.6, 0.1),
    (0.7, 0.3),
    (0.9, 0.2)
])

plt.show()
```

Python scripts and modules

A *module* is a file containing a collection of Python definitions and statements, typically named with a `.py` suffix.

A *script* is a module that is intended to be run by the Python interpreter. For example, the script `draw_triangles.py` can be run from the command-line using the command:

```
$ python draw_triangles.py
```

If you are using an Integrated Development Environment like Spyder or PyCharm, then the script can be run by opening it in the IDE and clicking on the "Run" button.

Modules, or specific functions from a module can be imported using the `import` statement:

```
import draw_triangles
from draw_triangles import draw_triangle
```

When a module is imported, all the statements in the module are executed by the Python interpreter. This happens only the first time the module is imported.

It is sometimes useful to have both importable functions as well as executable statements in a single module. When importing functions from this module, it is possible to avoid running other code by placing it under `if __name__ == "__main__"`:

Listing 3: draw_triangles.py

```
import matplotlib.pyplot as plt

def draw_triangle(points, ax=None):
    if ax is None:
        ax = plt.gca()
    else:
        fig, ax = plt.subplots()
        ax.set_xlabel('x')
        ax.set_ylabel('y')

    patch = plt.Polygon(points)
    ax.add_patch(patch)

    for pt in points:
        x, y = pt
```

(continues on next page)

(continued from previous page)

```
        ax.text(x, y, '({}, {})'.format(x, y))

if __name__ == "__main__":

    draw_triangle([
        (0.2, 0.2),
        (0.2, 0.6),
        (0.4, 0.4)
    ])

    draw_triangle([
        (0.6, 0.8),
        (0.8, 0.8),
        (0.5, 0.5)
    ])

    draw_triangle([
        (0.6, 0.1),
        (0.7, 0.3),
        (0.9, 0.2)
    ])

    plt.show()
```

When another module imports the module `draw_triangles` above, the code under `if __name__ == "__main__"` is **not** executed.

3.1.2 How to structure a Python project?

Let us now imagine we had a lot more code; for example, a *collection* of functions for:

- plotting shapes (like `draw_triangle` above)
- calculating areas
- geometric transformations

What are the different ways to organize code for a Python project that is more than a handful of lines long?

A single module

```
geometry
└── draw_triangles.py
```

One way to organize your code is to put all of it in a single `.py` file (module) like `draw_triangles.py` above.

Multiple modules

For a small number of functions the approach above is fine, and even recommended, but as the size and/or scope of the project grows, it may be necessary to divide up code into different modules, each containing related data and functionality.

```
geometry
├── draw_triangles.py
└── graphics.py
```

Listing 4: graphics.py

```
import matplotlib.pyplot as plt

def draw_triangle(points, ax=None):
    if ax is None:
        ax = plt.gca()
    else:
        fig, ax = plt.subplots()
        ax.set_xlabel('x')
        ax.set_ylabel('y')

    patch = plt.Polygon(points)
    ax.add_patch(patch)

    for pt in points:
        x, y = pt
        ax.text(x, y, '({}, {})'.format(x, y))
```

Typically, the "top-level" executable code is put in a separate script which imports functions and data from other modules:

Listing 5: draw_triangles.py

```
import graphics

graphics.draw_triangle([
    (0.2, 0.2),
    (0.2, 0.6),
    (0.4, 0.4)
])

graphics.draw_triangle([
    (0.6, 0.8),
    (0.8, 0.8),
    (0.5, 0.5)
])

graphics.draw_triangle([
    (0.6, 0.1),
    (0.7, 0.3),
    (0.9, 0.2)
])
```

Packages

A Python **package** is a directory containing a file called `__init__.py`, which can be empty. Packages can contain modules as well as other packages (sometimes referred to as *sub-packages*).

For example, `geometry` below is a package, containing various modules:

```
draw_triangles.py
geometry
├── graphics.py
└── __init__.py
```

A module from the package can be imported using the "dot" notation:

```
import geometry.graphics
geometry.graphics.draw_triangle(args)
```

It's also possible to import a specific function from the module:

```
from geometry.graphics import draw_triangle
draw_triangle(args)
```

Packages can themselves be imported, which really just imports the `__init__.py` module.

```
import geometry
```

If `__init__.py` is empty, there is "nothing" in the imported `geometry` package, and the following line gives an error:

```
geometry.graphics.draw_triangle(args)
```

```
AttributeError: module 'geometry' has no attribute 'graphics'
```

3.1.3 Importing from anywhere

sys.path

To improve their reusability, you typically want to be able to `import` your modules and packages from anywhere, i.e., from any directory on your computer.

One way to do this is to use `sys.path`:

```
import sys
sys.path.append('/path/to/geometry')

import graphics
```

`sys.path` is a list of directories that Python looks for modules and packages in when you `import` them.

Installable projects

A better way is to make your project "installable" using setuptools. To do this, you will need to include a `setup.py` with your project. Your project should be organized as follows:

```
draw_triangles.py
geometry
├── graphics.py
└── __init__.py
setup.py
```

A minimal `setup.py` can include the following

Listing 6: setup.py

```
from setuptools import setup

setup(name='geometry',
   version='0.1',
   author='Ashwin Srinath',
   packages=['geometry'])
```

You can install the package using `pip` with the following command (run from the same directory as `setup.py`):

```
$ pip install -e . --user
```

This installs the package in *editable* mode, creating a link to it in the user's `site-packages` directory, which happens to already be in `sys.path`.

Once your project is installed, you don't need to worry about adding it manually to `sys.path` each time you need to use it.

It's also easy to *uninstall* a package; just run the following command from the same directory as `setup.py`:

```
$ pip uninstall .
```

3.2 Testing your code

Note: This section is based heavily on Ned Batchelder's excellent article and PyCon 2014 talk Getting Started Testing.

Tests are the dental floss of development: everyone knows they should do it more,
but they don't, and they feel guilty about it.
- Ned Batchelder

Code without tests should be approached with a 10-foot pole.
- me

How can you write modular, extensible, and reusable code?

After making changes to a program, how do you ensure that it will still give the same answers as before?

How can we make finding and fixing bugs an easy, fun and rewarding experience?

These seemingly unrelated questions all have the same answer, and it is **automated testing**.

3.2.1 Testing by example: `flip_string`

Here is a function called `flip_string` that flips (reverses) a string. There are bug(s) in this function that we need to find and fix. Test the function for various inputs and compare the results obtained with expected output.

Listing 7: flip_string.py

```
def flip_string(s):
    """
    flip_string: Flip a string

    Parameters
    ----------
    s : str
        String to reverse

    Returns
    -------
    flipped : str
        Copy of `s` with characters arranged in reverse order
    """

    flipped = ''

    # Starting from the last character in `s`,
    # add the character to `flipped`,
    # and proceed to the previous character in `s`.
    # Stop whenever we reach the first character.

    i = len(s)

    while True:
        i = i-1
        char = s[i]
        flipped = flipped + char

        # stop if we have reached the first character:
        if char == s[0]:
           break

    return flipped
```

- What tests did you come up with? Why did you choose those tests?
- How did you organize and execute your tests?
- Can the results of your tests help you figure out what problem(s) there might be with the code?

Testing interactively

This is the most common type of testing, and something you have probably done before. To test a function or a line of code, you simply fire up an interactive Python interpreter, import the function, and test away:

```
>>> from flip_string import flip_string
>>> flip_string('mario')
'oiram'
>>> flip_string('luigi')
'igiul'
```

While this kind of testing is better than not doing any testing at all, it leaves much to be desired. First, it needs to be done each time `flip_string` is changed. It also requires that we manually inspect the output from each test to

decide if the code "passes" or "fails" that test. Further, we need to remember all the tests came up with today if we want to test again tomorrow.

Writing a test script

A *much* better way to write tests is to put them in a script:

Listing 8: test_flip_string.py

```
from flip_string import flip_string

flipped = flip_string("mario")
print("mario flipped is:", flipped)

flipped = flip_string("luigi")
print("luigi flipped is:", flipped)
```

Now, running and re-running our tests is very easy - we just run the script:

```
$ python test_flip_string.py
mario flipped is: oiram
luigi flipped is: igiul
```

It's also easy to add new tests, and there's no need to remember all the tests we come up with.

Testing with assertions

One problem with the method above is that we *still* need to manually inspect the results of our tests.

Assertions can help with this.

The `assert` statement in Python is very simple: Given a condition, like `1 == 2`, it checks to see if the condition is true or false. If it is true, then `assert` does nothing, and if it false, it raises an `AssertionError`:

```
>>> assert 1 == 1
>>> assert 1 < 2
>>> assert 1 > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

We can re-write our script `test_flip_string.py` using assertions as follows:

Listing 9: test_flip_string.py

```
from flip_string import flip_string

assert flip_string('mario') == 'oiram'
assert flip_string('luigi') == 'igiul'
```

And we still run our tests the same way:

```
$ python test_flip_string.py
```

This time, there's no need to inspect the test results. If we get an `AssertionError`, then we had a test fail, and if not, all our tests passed.

However, there's no way to know if *more* than one test failed. The script stops executing after the first `AssertionError` is encountered.

Let's add another test to our test script and re-run it:

Listing 10: test_flip_string.py

```
from flip_string import flip_string

assert flip_string('mario') == 'oiram'
assert flip_string('luigi') == 'igiul'
assert flip_string('samus') == 'sumas'
```

```
$ python test_flip_string.py

Traceback (most recent call last):
  File "test_flip_string.py", line 5, in <module>
    assert flip_string('samus') == 'sumas'
AssertionError
```

This time we get a failed test, because - as we said - our code has bugs in it. Before adding more tests to investigate further, we'll discuss one more method for running tests.

Using a test runner

A test runner takes a bunch of tests, executes them all, and then reports which of them passed and which of them failed.

A very popular test runner for Python is pytest.

To run our tests using pytest, we need to re-write them as follows (essentially, wrap each test in a function):

Listing 11: test_flip_string.py

```
from flip_string import flip_string

def test_flip_mario():
    assert flip_string('mario') == 'oiram'

def test_flip_luigi():
    assert flip_string('luigi') = 'igiul'

def test_flip_samus():
    assert flip_string('samus') == 'sumas'
```

To run our tests, we simply type `pytest` on the command line. When we do this, pytest will look for all files containing tests, run all the tests in those files, and report what it found:

```
$ pytest

collected 3 items

test_flip_string.py ..F                                              [100%]

================================== FAILURES ===================================
_____________________________ test_flip_samus _____________________________
```

(continues on next page)

(continued from previous page)

```
    def test_flip_samus():
>       assert flip_string('samus') == 'sumas'
E       AssertionError: assert 's' == 'sumas'
E         - s
E         + sumas

test_flip_string.py:10: AssertionError
====================== 1 failed, 2 passed in 0.07 seconds ======================
```

As you can see above, pytest prints a lot of useful information in its report. First, it prints a summary of passed v/s failed tests:

```
test_flip_string.py ..F                                                  [100%]
```

A dot (`.`) indicates a passed test, while a `F` indicates a failed test.

For each failed test, it provides further information, including the expected value as well as the obtained value in the failed assertion:

```
================================= FAILURES =================================
_____________________________ test_flip_samus ______________________________

    def test_flip_samus():
>       assert flip_string('samus') == 'sumas'
E       AssertionError: assert 's' == 'sumas'
E         - s
E         + sumas

test_flip_string.py:10: AssertionError
```

Useful tests

Now that we know how to write and run tests, what kind of tests should we write? Testing `flip_string` for arbitrary words like `'mario'` and `'luigi'` might not tell us much about where the problem might be.

Instead, we should choose tests that exercise specific functionality of the code we are testing, or represent different conditions that the code may be exposed to.

Here are some examples of more useful tests:

- Flipping a string with a single character (no work needs to be done)
- Flipping a string with two characters (minmum amount of work needs to be done)
- Flipping a string that reads the same forwards and backwards

Listing 12: test_flip_string.py

```
from flip_string import flip_string

def test_flip_one_char():
    assert flip_string('a') == 'a'

def test_flp_two_charsi():
    assert flip_string('ab') == 'ba'
```

(continues on next page)

(continued from previous page)

```
def test_flip_palindrome():
    assert flip_string('aba') == 'aba'
```

```
collected 3 items

test_flip_string-v5.py ..F                                               [100%]

================================== FAILURES ===================================
_____________________________ test_flip_palindrome _____________________________

    def test_flip_palindrome():
>       assert flip_string('aba') == 'aba'
E       AssertionError: assert 'a' == 'aba'
E         - a
E         + aba

test_flip_string.py:10: AssertionError
====================== 2 failed, 1 passed in 0.08 seconds ======================
```

Fixing the code

From the test results above, we see that `flip_string` failed for the input `'aba'`. Now, can you trace the execution of the code in the function `flip_string` for this input and figure out why it returned `a`?

After fixing the code, re-run the tests to make sure you didn't break anything else in the process of fixing this bug – this is one of the reasons tests are so valuable!

3.2.2 Types of testing

Software testing is a vast topic and there are many levels and types of software testing.

For scientific and research software, the focus of testing efforts is primarily:

1. **Unit tests**: Unit tests aim to test small, independent sections of code (a function or parts of a function), so that when a test fails, the failure can easily be associated with that section of code. This is the kind of testing that we have been doing so far.
2. **Regression tests**: Regression tests aim to check whether changes to the program result in it producing different results from before. Regression tests can test larger sections of code than unit tests. As an example, if you are writing a machine learning application, you may want to run your model on small data in an automated way each time your software undergoes changes, and make sure that the same (or a better) result is produced.

3.2.3 Test-driven development

Test-driven development (TDD) is the practice of writing tests for a function or method *before* actually writing any code for that function or method. The TDD process is to:

1. Write a test for a function or method
2. Write just enough code that the function or method passes that test
3. Ensure that all tests written so far pass
4. Repeat the above steps until you are satisfied with the code

Proponents of TDD suggest that this results in better code. Whether or not TDD sounds appealing to you, writing tests should be *part* of your development process, and never an afterthought. In the process of writing tests, you often come up with new corner cases for your code, and realize better ways to organize it. The result is usually code that is more modular, more reusable and of course, more testable, than if you didn't do any testing.

3.2.4 Growing a useful test suite

More tests are always better than less, and your code should have as many tests as you are willing to write. That being said, some tests are more useful than others. Designing a useful suite of tests is a challenge in itself, and it helps to keep the following in mind when growing tests:

1. **Tests should run quickly**: testing is meant to be done as often as possible. Your entire test suite should complete in no more than a few seconds, otherwise you won't run your tests often enough for them to be useful. Always test your functions or algorithms on very small and simple data; even if in practice they will be dealing with more complex and large datasets.
2. **Tests should be focused**: each test should exercise a small part of your code. When a test fails, it should be easy for you to figure out which part of your program you need to focus debugging efforts on. This can be difficult if your code isn't modular, i.e., if different parts of your code depend heavily on each other. This is one of the reasons TDD is said to produce more modular code.
3. **Tests should cover all possible code paths**: if your function has multiple code paths (e.g., an if-else statement), write tests that execute both the "if" part and the "else" part. Otherwise, you might have bugs in your code and still have all tests pass.
4. **Test data should include difficult and edge cases**: it's easy to write code that only handles cases with well-defined inputs and outputs. In practice however, your code may have to deal with input data for which it isn't clear what the behaviour should be. For example, what should `flip_string('')` return? Make sure you write tests for such cases, so that you force your code to handle them.

3.3 Documenting your code

Most people think of writing documentation as an unpleasant, but necessary task, done for the benefit of othe people with no real benefit to themselves. So they choose not to do it, or they do it with little care.

But even if you are the only person who will ever use your code, it's still a good idea to document it well. Being able to document your own code gives you confidence that you understand it yourself, and a sign of well-written code is that it can be easily documented. Code you wrote a few weeks ago may as well have been written by someone else, and you will be glad that you documented it.

The good news is that writing documentation can be fun, and you really don't need to write a lot of it.

3.3.1 Docstrings and comments

Documentation is *not* comments.

A *docstring* in Python is a string literal that appears at the beginning of a module, function, class, or method.

```
"""
A docstring in Python that appears
at the beginning of a module, function, class or method.
"""
```

The *docstring* of a module, function, class or method becomes the `__doc__` attribute of that object, and is printed if you type `help(object)`:

```
In [1]: def fahr_to_celsius(F):
   ...:     """
   ...:     Convert temperature from Fahrenheit to Celsius.
   ...:     """
   ...:     return (F - 32) * (5/9)

In [2]: help(fahr_to_celsius)

Help on function fahr_to_celsius in module __main__:

fahr_to_celsius(F)
 Convert temperature from Fahrenheit to Celsius.
```

A *comment* in Python is any line that begins with a #:

```
# a comment.
```

The purpose of a docstring is to document a module, function, class, or method. The purpose of a comment is to explain a very difficult piece of code, or to justify a choice that was made while writing it.

Docstrings should not be used in place of comments, or vice versa. **Don't do the following**:

```
In [1]: def fahr_to_celsius(F):
   ...:     # Convert temperature from Fahrenheit to Celsius.
   ...:     return (F - 32) * (5/9)
```

Deleting code

Incidentally, many people use comments and string literals as a way of "deleting" code - also known as *commenting out* code. See this article on a better way to delete code.

3.3.2 What to document?

So what goes in a dosctring?

At minimum, the docstring for a function or method should consist of the following:

1. A **Summary** section that describes in a sentence or two what the function does.
2. A **Parameters** section that provides a description of the parameters to the function, their types, and default values (in the case of optional arguments).
3. A **Returns** section that similarly describes the return values.
4. Optionally, a **Notes** section that describes the implementation, and includes references.

Here is a simple example of this in action:

```
def flip_list(a, inplace=False):
    """
    Flip (reverse) a list.

    Parameters
    ----------
    a : list
        List to be reversed.
    inplace : bool, optional
```

(continues on next page)

(continued from previous page)

```
        Specifies whether to flip the list "in place",
        or return a new list (default).

    Returns
    -------
    flipped : list (or None)
        The flipped list. If `inplace=True`, None is returned.
    """
    if inplace is True:
        a[:] = a[::-1]
        return None
    else:
        return a[::-1]
```

NumPy's documentation guidelines are a great reference for more information about what and how to document your code.

3.3.3 Doctests

In addition to the sections above, your documentation can also contain runnable tests. This is possible using the doctest module.

Listing 13: flip_list.py

```
def flip_list(a, inplace=False):
    """
    Flip (reverse) a list.

    Parameters
    ----------
    a : list
        List to be reversed.
    inplace : bool, optional
        Specifies whether to flip the list "in place",
        or return a new list (default).

    Returns
    -------
    flipped : list (or None)
        The flipped list. If `inplace=True`, None is returned.


    >>> flip_list([1, 2, 3])
    [3, 2, 1]

    >>> a = [1, 2, 3]
    >>> flip_list(a, inplace=True)
    >>> a
    [3, 2, 1]
    """
    if inplace is True:
        a[:] = a[::-1]
        return None
    else:
        return a[::-1]
```

You can tell `pytest` to run doctests as well as other tests using the `--doctest-modules` switch:

```
$ pytest --doctest-modules flip_list.py

collected 1 item

flip_list.py .                                                           [100%]

========================== 1 passed in 0.03 seconds ==========================
```

Doctests are great because they double up as documentation as well as tests. But they shouldn't be the *only* kind of tests you write.

3.3.4 Documentation generation

Finally, you can turn your documentation into a beautiful website (like this one!), a PDF manual, and various other formats, using a document generator such as Sphinx. You can use services like readthedocs to build and host your website for free.

3.4 Improving the usability of Python programs

3.4.1 Logging

It can be useful to print out either a message or the value of some variable, etc., while your code is running. This is quite common and is usually accomplished with a simple call to the `print` function.

```
x = 1.234
print("The value of x is {0:0.4f}.".format(x))
```

```
The value of x is 1.2340.
```

Doing this is a good idea to keep track of milestones in your code. That way, both when you are developing your code but also when other users are running the code, they can be notified of an event, progress, or value.

Printing a message is also useful for notifying the user when something is not going as expected. These are all different *levels* of messaging.

Logging is simply engaging in this behavior of printing out messages, with the added feature that you include meta data (e.g., a timestamp, the message category) with the message, as well as a filter where only messages with a high enough level of criticality are actually allowed to be printed.

Logging Basics

The general idea is that there are multiple levels of messages that can be printed. Typically these include:

1. DEBUG - diagnostic purposes.
2. INFO - basic information (most common).
3. WARNING - indicating non-normal behavior.
4. ERROR - error (the operation cannot continue).
5. CRITICAL - error (the program cannot continue).

During the initialization portion of your code, you would configure a *logger* object with a format, where to print messages (e.g., console, file, or both), and what level to use by default. Usually, you would set the default log level to `INFO` and the debugging messages used for diagnostics would not actually be printed. Then, allow the user to override this with a *command line argument* (e.g., `--debug`).

Example Setup

Python has a logging module as part of the standard library. It is very comprehensive and allows the user to heavily customize many parts of the behavior. It is pretty strait forward to implement your own logging functionality; unless you're doing something special why not use the standard library?

```
import logging

log = logging.getLogger("ProjectName")

file_handler = logging.FileHandler("path/for/output.log")
console_handler = logging.StreamHandler()

formatter = logging.Formatter("%(levelname)s %(asctime)s %(name)s - %(message)s")
file_handler.setFormatter(formatter)
console_handler.setFormatter(formatter)

log.addHandler(file_handler)
log.addHandler(console_handler)
log.setLevel(logging.INFO)
```

Then, somewhere in the code:

```
log.debug("report on some variable")
log.info("notification of milestone")
log.warn("non-standard behavior")
log.error("unrecoverable issue")
log.critical("panic!")
```

```
INFO 2018-07-24 09:41:56,683 ProjectName - notification of milestone
WARNING 2018-07-24 09:41:56,835 ProjectName - non-standard behavior
ERROR 2018-07-24 09:41:57,103 ProjectName - unrecoverable issue
CRITICAL 2018-07-24 09:41:57,103 ProjectName - panic!
```

Notice that the debug message was not printed. This is because we set the log level to `INFO`. Only messages with a level equal to or higher then the assigned level will make it passed the filter.

Logging with Color

Finally, another common feature of logging is to add color as an indicator of the message type. Obviously, this only applies to messages that are printed to the console. If you've ever started up a *Jupyter* notebook server you might have noticed the logging messages it puts out a similar format as used here and the meta data is a bold color. The color codes are generally as follows:

- DEBUG (blue)
- INFO (green)
- WARNING (orange or yellow)
- ERROR (red)

- CRITICAL (purple)

3.4.2 Command Line Arguments

In addition to packaging your code in a way that other users or projects can import for use in their code, often it makes sense to also make elements of the code executable from the command line as stand alone scripts. Python has everything you need to do this built right in.

As with logging, there are several python packages available that handle command line argument parsing for you, including a robust implementation provided right in the standard library - argparse.

The *argparse* module, as well as the others, rely on a universally excepted convention for how command line arguments should be structured. Nearly all of the standard utilities on Unix/Linux systems use this same syntax. This convention covers both the command line argument syntax as well as the structure of *usage* statements that your script prints out (e.g., when supplying the `--help` option). The *argparse* module actually takes care of all of this for you.

Unix Convention

There is a fair bit of complexity to the convention surrounding the *usage* statements, but the argument syntax is fairly simple.

Positional arguments are those that don't have names. These are usually file paths in the context of analysis scripts. *Optional arguments* are those that have defaults and may or may not accept a value.

Optional arguments can be specified with *short form* or *long form* names (usually both). The short form names are a single letter preceded by a single dash (e.g., `-a`). Short form options that don't take an argument can be stacked (e.g., `-abc`). Long form arguments are whole words and preceded by two dashes (e.g., `--debug`). Long form arguments that are multiple words are usually joined with dashes (e.g., `--output-directory`).

There is more, but these are the basics.

Simple Example

The best (most robust and cross-platform) way of providing a stand along script with your package is to let your *setup.py* file handle it. Doing the following will create the proper executable on both Windows and Unix systems and put it in a place that is readily callable (i.e., on the user's *PATH*).

```
# setup.py

# use "entry_points" to point to function and setuptools
# will create executables on your behalf.
setup(
# ...
    # syntax: "{name}={package}.{module}:{function}"
    # "{name}" will be on your PATH in the same "/bin/"
    # alongside python/pip executables.
    entry_points = {"console_scripts": [
        "do_science=my_package.do_science:main",
    ]},
# ...
)
```

This says that I have a file, `my_package/do_science.py`, with a function called `main` that when called does the thing I want the script to do. The function won't be given any arguments, but we can get what we need from `sys.argv`. This has the effect of creating an executable we can invoke with the name `do_science` that behaves equivalent to the following.

```
import sys
from my_package.do_science import main
sys.exit(main())
```

With this in mind, your function can and should return integer values which will be used as the exit status of the command. This is another Unix convention; returning zero is for success, returning a non-zero status indicates some specific error has occurred.

The following shows a basic usage of `argparse` and how to define your "main" function.

```
# do_science.py
# script for doing cool science things

import argparse

parser = argparse.ArgumentParser(prog="do_science",
                                 description="do cool science thing")

# positional argument
parser.add_argument("input_file", help="path to input data file")

# optional argument
parser.add_argument("-d", "--debug", action="store_true",
                    help="enable debugging messages")

def main() -> int:
    """Main entry point for `do_science`.

       Returns:
       exit_status: int
           0 if success, non-zero otherwise.
    """

    # parse_args() automatically grabs sys.argv if you don't provide them.
    opts = parser.parse_args()
    # opts is a namespace
    # opts.input_file is a string with the value from the command line
    # opts.debug is True or False (default is False w/ "store_true")
    return 0
```

After the package is installed, `pip install my_package ...`, you'll be able to call the script:

```
> do_science
usage: do_science [-h] [-d] input_file
```

```
> do_science --help
usage: do_science [-h] [-d] input_file

do cool science thing

positional arguments:
  input_file   path to input data file

optional arguments:
  -h, --help   show this help message and exit
  -d, --debug  enable debugging messages
```

3.5 Improving the performance of Python programs

3.5.1 Timing code and identifying bottlenecks

Of course, the first step toward improving performance is to figure out where to focus your efforts. This means identifying the section of code in your program that is taking the most time, i.e., the "bottleneck".

Sometimes, the bottleneck is very obvious (e.g., the training step in a machine learning application), and sometimes it may not be clear. In the latter case, you need to be able to measure the time taken by various parts of your program.

The `time` function

The time function can be used to time a section of code as follows:

```
import time
import numpy as np

t1 = time.time()
a = np.random.rand(5000, 5000)
t2 = time.time()
print("Generating random array took {} seconds".format(t2-t1))
```

```
Generating random array took 0.44880104064941406 seconds
```

`%timeit` and `%%timeit`

`%timeit%` and `%%timeit` are magic statements that can be used in IPython or in Jupyter Notebook for timing a single line of code or a block of code conveniently:

```
In [1]: import numpy as np

In [2]: %timeit np.random.rand(5000, 5000)
410 ms ± 2.59 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [3]: %%timeit
   ...: a = np.random.rand(5000, 5000)
   ...: b = np.random.rand(5000, 5000)
   ...: c = a * b
   ...:
897 ms ± 10.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Profilers

`time` and `timeit` should help with most of your measurement needs, but if you need to profile a very long program with lots of functions, you may benefit from using a profiler.

There is also a line_profiler that can help you automatically profile each line in a script, and a memory_profiler to measure memory consumption.

3.5.2 Install optimized versions of libraries

This is the easiest way to get "free" performance improvements. If your computer supports it, install optimized version of Python libraries, for example, those provided by the Intel Distribution for Python.

Another option is PyPy.

3.5.3 Choose the right algorithm

This is one of the most effective ways to improve the performance of a program.

When choosing a function from a library or writing your own, ensure that you understand how it will perform for the type and size of data you have, and what options there may be to boost its performance. Always benchmark to compare with other functions and libraries.

For example, if you are doing linear algebra, you may benefit from the use of sparse matrices and algorithms if you are dealing with very large matrices with relatively few non-zeros.

As another example, many kinds of algorithms are iterative and require an initial "guess" for the solution. Typically, the closer this initial guess is to the actual solution, the faster the algorithm performs.

3.5.4 Choose the appropriate data format

Familiarize yourself with the various data formats available for the type of data you are dealing with, and the performance considerations for each. For example, this page provides a good overview of various data formats for tabular data supported by the Pandas library. Performance for each is reported here.

3.5.5 Don't reinvent the wheel

Resist any temptation to write your own implementation for a common task or a well-known algorithm. Rely instead on other well-tested and well-used implementations.

For instance, it's easy to write a few lines of Python to read data from a `.csv` file into a Pandas DataFrame:

Listing 14: my_csv.py

```
def read_csv(fname):
    with open(fname) as f:
        col_names = f.readline().rstrip().split(',')
        df = pandas.DataFrame(columns=col_names)
            for line in f:
                record = pandas.DataFrame([line.rstrip().split(',')], columns=col_
↪names)
                df = df.append(record, ignore_index=True)
    return df
```

But such code performs poorly. Compare the performance with Pandas' `read_csv` function:

```
In [1]: from my_csv import read_csv

In [2]: %time data = read_csv('feet.csv')
CPU times: user 2min 3s, sys: 1.39 s, total: 2min 4s
Wall time: 2min 5s
```

```
In [1]: from pandas import read_csv

In [2]: %time data = read_csv('feet.csv')
CPU times: user 28.5 ms, sys: 10.8 ms, total: 39.3 ms
Wall time: 54.2 ms
```

It also isn't nearly as versatile, and doesn't account for the dozens of edge cases than Pandas does.

3.5.6 Benchmark, benchmark, benchmark!

If there are two ways of doing the same thing, *benchmark* to see which is faster for different problem sizes.

For example, let's say we want to compute the average `hindfooth_length` for all species in `plot_id` 13 in the following dataset:

```
In [1]: data = pandas.read_csv('feet.csv')

In [2]: data.head()
Out[2]:
   plot_id species_id  hindfoot_length
0        2         NL             32.0
1        3         NL             33.0
2        2         DM             37.0
3        7         DM             36.0
4        3         DM             35.0
```

One way to do this would be to group by the `plot_id`, compute the mean hindfoot length for each group, and extract the result for the group with `plot_id` 13:

```
In [2]: data.groupby('plot_id')['hindfoot_length'].mean()[13]
Out[2]: 27.570887035633056
```

Another way would be to filter the data first, keeping only records with `plot_id` 13, and then computing the mean of the `hindfoot_length` column:

```
In [3]: data[data['plot_id'] == 13]['hindfoot_length'].mean()
Out[3]: 27.570887035633056
```

Both methods give identical results, but the difference in performance is significant:

```
In [4]: %timeit data.groupby('plot_id')['hindfoot_length'].mean()[13]
1.34 ms ± 24.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

In [5]: %timeit data[data['plot_id'] == 13]['hindfoot_length'].mean()
750 µs ± 506 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Why do you think the first method is slower?

3.5.7 Avoid explicit loops

Very often, you need to operate on multiple elements of a collection such as a NumPy array or Pandas DataFrame.

In such cases, it is almost always a bad idea to write an explicit `for` loop over the elements.

For instance, looping over the rows (a.k.a, *indices* or *records*) of a Pandas DataFrame is considered poor practice, and is very slow. Consider replacing values in a column of a dataframe:

```
In [5]: %%timeit
   ...: for i in range(len(data['species_id'])):
   ...:     if data.loc[i, 'species_id'] == 'NL':
   ...:         data.loc[i, 'species_id'] = 'NZ'
   ...:
308 ms ± 4.49 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

A better way to do this is simply to use the `replace()` method:

```
In [2]: %time data['species_id'].replace('NL', 'NZ', inplace=True)
CPU times: user 3.1 ms, sys: 652 µs, total: 3.75 ms
Wall time: 3.34 ms
```

In addition to being faster, this also leads to more readable code.

Of course, loops are unavoidable in many situations; but look for alternatives before you write a `for` loop over the elements of an array, DataFrame, or similar data structure.

3.5.8 Avoid repeatedly allocating, copying and rearranging data

Repeatedly creating and destroying new data can be very expensive especially if you are working with very large arrays or data frames. So avoid, for instance, creating a new array each time inside a loop. When operating on NumPy arrays, memory is allocated for intermediate results. Packages like numexpr aim to help with this.

Understand when data needs to be copied v/s when data can be operated "in-place". It also helps to know *when* copies are made. For example, do you think the following code results in two copies of the same array?

```
import numpy as np

a = np.random.rand(50, 50)
b = a
```

This article clears up a lot of confusion about how names and values work in Python and when copies are made v/s when they are not.

3.5.9 Access data from memory efficiently

Accessing data in the "wrong order": it is always more efficient to access values that are "closer together" in memory than values that are farther apart. For example, looping over the elements along the rows of a 2-d NumPy array is *much* more efficient than looping over the elements along its columns. Similarly, looping over the columns of a DataFrame in Pandas will be faster than looping over its rows.

- Redundant computations / computing "too much": if you only need to compute on a subset of your data, filter *before* doing the computation rather than after.

3.5.10 Interfacing with compiled code

You may have heard that Python is "slow" compared to other languages like C, C++, or Fortran. This is somewhat true in that Python programs written in "pure Python", i.e., without the use of any libraries except the standard libraries, will be slow compared to their C/Fortran counterparts. One of the reasons that C is so much faster than Python is that it is a compiled language, while Python is an interpreted language.

However, the core of libraries like NumPy are actually written in C, making them much faster than "pure Python".

It's also possible for you to write your own code so that it interfaces with languages like C, C++ or Fortran. Better still, you often don't even need to write any code in those languages, and instead can have other libraries "generate" them for you.

Numba is a library that lets you compile code written in Python using a very convenient "decorator" syntax.

As an example, consider numerically evaluating the derivative of a function using finite differences. A function that uses NumPy to do this might look like the following:

Listing 15: derivatives.py

```
import numpy as np

def dfdx(f, dx, y):
   y[1:-1] = (f[2:] - f[:-2]) / (2*dx)
   y[0] = (f[1] - f[0]) / dx
   y[1] = (f[-2] - f[-1]) / dx
   return y
```

Below, we time the function for a grid of 10000000 points:

```
In [1]: x = np.linspace(0, 1, 10000000)

In [2]: dx = x[1] - x[0]

In [3]: f = np.sin(2 * np.pi * x / 1000000)

In [4]: y = np.zeros_like(f)

In [5]: %timeit dfdx(f, dx, y)
61.1 ms ± 2.62 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Below is a function that is compiled using Numba to do the same task:

Listing 16: derivatives.py

```
from numba import jit, prange

@jit(parallel=True, nopython=True)
def dfdx(f, dx, y):
    for i in prange(1, len(y)-1):
        y[i] = (f[i+1] - f[i-1]) / 2*dx
    y[0] = (f[1] - f[0]) / dx
    y[-1] = (f[-1] - f[-2]) / dx
    return y
```

We see much better performance for the same grid size:

```
In [1]: %timeit dfdx(f, dx, y)
14.6 ms ± 282 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Cython is another option for interfacing with compiled code. It performs about the same as Numba but requires much more effort; although it can do many things that Numba cannot, such as generating C code, and interface with C/C++ libraries.

3.5.11 Parallelization

Finally, if your computer has multiple cores, or if you have access to a bigger computer (e.g., a high-performance computing cluster), parallelizing your code may be an option.

- Note that many libraries support parallelization without any effort on your part. Libraries like Numba and Tensorflow can use all the cores on your CPU, and even your GPU for accelerating computations.
- Dask is a great library for parallelizing computations and operating on large datasets that don't fit in RAM.
- The multiprocessing package is useful when you have several independent tasks that can all be done concurrently. joblib is another popular library for this.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search