Introduction to basic Python

Contents

Python is an astronomer's secret weapon. With Python, the process of visualizing, processing, and interacting with data is made extremely simple. Automatic data reduction, rapid edits to FITS headers, analysis of complex datasets, data mining of huge surveys–with a basic knowledge of Python, all of this will be within your grasp.

This document covers the fundamentals of Python, and is aimed at complete beginners. It does **not** need to be read in one sitting; in fact, you are encouraged to come back to it again and again over a period of weeks, as you learn more about Python.

A word about practice

As soon as you begin learning, you should also start practicing by writing your own code. You may be surprised at how quickly real-world practice turns concepts which seemed strange and difficult into second nature. And each time this happens, you will think more and more like a programmer, so that your learning becomes easier with time.

Installing Python

1. Download the Python installer from https://www.python.org/downloads/
2. Launch the installer
3. **VERY IMPORTANT: Make sure that the option "Add Python to PATH" is checked!** This is what tells your computer where to find Python when you need it.
4. Click "Install Now"

Once the installer is finished, it's time to test whether the installation worked:

1. Open your computer's terminal window. (On Windows, the terminal is known as the Command Prompt. On Mac, it's just called Terminal.)
2. Run this command: `py --version`

If you see a message like "`Python 3.10.6`", that means Python was installed successfully. Congratulations!

If an error shows up instead, that means Python was not installed successfully, and/or your computer doesn't know where to find it. Uninstall Python, and try again.

How to run Python code

If you're new to programming, you might be wondering exactly how you're supposed to run your code once you've written it. This page will show you how.

Remember: Python is just a program

"Wait just a minute", I hear you saying. "Isn't Python a programming *language*? The stuff that programs are made out of? How can it be a program itself?"

Python is an **interpreted language**. What that means is that computers can't run Python code by default, so you need to download and install a program called an "interpreter". That's what you're actually doing when you install Python.

The Python interpreter reads text (written by you) and converts it to other types of code which *can* run on a computer. (This extra conversion step makes Python slower than some other programming languages, but that's fine since we're not trying to write super fast 3D video games, just process astronomical data.)

The Python interpreter is a piece of software, just like a web browser or a planetarium program. The only difference is that the Python interpreter is invisible– it doesn't have a GUI window for you to click on. But it's still a piece of software which can be installed, run, updated, and uninstalled.

Method #1: Writing code in real time

The simplest way to run small amounts of Python code is to use the interpreter in real time from a terminal window.

Here's how to launch the Python interpreter's interactive mode:

1. **Open your computer's terminal window.** On Windows, the terminal is known as the Command Prompt. On Mac, it's just called Terminal.
2. **Run this command:** `py`

You should now see three greater-than signs `>>>` followed by a blinking cursor. Whenever you see the marker `>>>`, that means you can type in a line of code. Hit `Enter` to run the code you just wrote, and move on to the next line.

What happens if I made a mistake?

If you make a mistake, it will raise an exception like normal, but it won't crash the entire program– instead, you will have the option to keep going where you left off.

How do I close it when I'm done?

Once you're done running code, you can close the Python interpreter using the `exit()` function. (Or, you can just close the terminal window.)

Method #2: Running a script file

You can also run Python code which has been written down in a text file (a.k.a. a "Python script"). This is how almost all Python programs are made.

Here's how to create a script file and run it:

1. **Make a new text file, and give it a name.** Make sure that the name ends with .py, instead of .txt.
2. **Write your code inside the text file.** It doesn't have to be 100% perfect to start out–you can always edit it later.
3. **Save your script.**
4. **Open your computer's terminal window.** On Windows, the terminal is known as the Command Prompt. On Mac, it's just called Terminal.
5. **"Mount" (navigate to) the folder which has your script file in it.** On Windows, if you saved your script file to your desktop, the command will probably look something like this: `cd C:\Users\YourUsernameHere\Desktop`
6. **Run this command:** `py myscript.py` (replace myscript.py with the name of your script file)

The Python interpreter will read your script file, convert it to code which your computer can understand, and run it.

What happens if I made a mistake?

If you make a mistake, it will raise an exception, and the script will crash. If that happens, don't panic–it's actually completely normal for your script to crash when you've just made changes to it. The traceback created by the exception will have helpful info in it to help improve your program.

You can edit your script file, save the changes, and run it again using the same command: `py myscript.py`. No need to retype the whole thing.

How do I close it when I'm done?

When it reaches the end of the script file, the Python interpreter knows that there won't be any more code coming, and closes itself. You don't have to do anything.

Which method should I use?

Method #1, the "one-line-at-a-time" interactive shell, is good for when you only need to run a few lines of code. An example would be if you want to rename all of the files in a directory, or open a single FITS file to check its header.

But since the interactive shell runs every line of code as soon as you enter it, if you make a mistake or forget to include something, you might have to just start over from the beginning– not fun! Plus, your code isn't saved anywhere, so if you want to run it again in the future, you'll have to rewrite it from scratch.

For these reasons, if you have more than 5-10 lines of code to run, you should consider using Method #2, and writing a script file instead.

With Method #2, all of your code is stored safely inside a script file. If you need to make a tiny change to line #3, you can do that without hurting any of the other lines of code. And since you can save your progress, you can also write really long programs, the kind that include hundreds of lines of code and take multiple days to figure out.

How to write Python code

In this section, we'll go through a dictionary-style introduction to Python. Here are a few tips for using this document to learn:

1. If you come across a word you don't know, try pressing `ctrl`+`f` to search this document– it might be covered elsewhere.
2. Don't be afraid to jump around. The concepts described below are sorted *roughly* by difficulty, but they're all highly related, so there's really no "best order" to learn them.

3. As you read about each concept, try testing them out in the terminal. When it comes to programming, you'll learn far faster from casual experimentation than you will reading!

Indentation

If you've ever used a different programming language before, you might remember using lots of curly braces, semicolons, and other markers in order to tell the computer where a piece of code started and ended.

As an example, here is one way to display a list of items using JavaScript:

```
for (let i = 0; i < MyArr.length; i++) {
    console.log(MyArr[i]);
}
```

Python can figure out where the different parts of code start and end just by looking at the whitespace, or indentation. This makes Python code **much** easier to read and understand.

Here is the same code as above, but written in Python:

```
for item in my_arr:
    print(item)
```

Much clearer!

If you ever mess up the indentation of your code, Python will tell you what went wrong by throwing an IndentationError exception:

```
for item in my_arr:
print(item)
```

```
  Input In [3]
    print(item)
    ^
IndentationError: expected an indented block
```

Python tells you that it expected to find a indented block, and displays a little arrow pointing at the place where it thinks the whitespace should go.

Comments

Comments are notes which start with a pound sign #. When the Python interpreter is reading your code, it will ignore comments completely, so you can write whatever you want in them. They're great for writing memos to help you remember why you wrote your code the way you did.

```
# This is a comment.

# Comments are useful for leaving notes inside your code.

'''This is a special type of string, which lots of people
use for long comments. The nice thing about these strings
is that you can include lots of line breaks, and the comment
won't end until you type another three single quotes.'''

"""You can use double quotes, too."""

# You can write a comment on the same line as some code,
x = 10 # as long as it's the last thing on the line.
```

Variables

Variable are used to store data, so that you can access it again later in your code. There are two basic types of variables: numbers and strings. (Strings are used to store text.)

```
pixel_size = 9           #number
read_noise = 3.425       #number
camera_name = 'my SBIG'  #string
observer_code = "HLAA"   #string
```

To access a variable's data, you can type its name:

```
camera_name
```

(A variable's name is used to tell *you*, the programmer, what's inside the variable. Python itself doesn't read or understand variable names, so you can make them say whatever you want.)

You can update a variable by overwriting it:

```
pixel_size = 2.75 #Changes the pixel size to 2.75
pixel_size = 2 * pixel_size #Changes the pixel size to 5.5

# All strings come with built-in methods for
# making simple changes, like capitalization.
camera_name = camera_name.capitalize()
```

Collections

Collections are variables that can hold more than one data point. There are four types of collection in Python: lists, dictionaries, tuples, and sets.

```
# Lists are the most common type of collection, and can be modified
easily.
filters = ['B','V','R','I']

# Dictionaries label each value with a unique 'key'.
# This makes it easy to 'look up' values later.
target = {'Name':'Betelgeuse', 'RA':88.8, 'DEC':7.4}

# Tuples are like lists, but they can't be modified after they're
created.
# You will probably not use them often.
coefficients = (9.993, 0.012, 0.4501)

# Sets can only store unique values.
# You'll probably never write one down like this, but you might
sometimes
# convert a list to a set and back to remove duplicate values.
observer_codes = {'HLAA', 'BSK'}
```

Collections can hold other variables, and even other collections:

```
camera = {
    'Name':camera_name,
    'Filters':filters,
    'Sensor size':[3064, 2048],
}
```

If you have two lists that you want to turn into a dictionary, you can do that by using the built-in zip() function to group them together first:

```
names = ['V1405 Cas', 'V1674 Her', 'RS Oph']
magnitudes = [5.2, 6.0, 4.3]

peak_magnitudes = dict(zip(names, magnitudes))
```

Lists and dictionaries are “mutable”, meaning that it’s possible to modify just part of them:

```
peak_magnitudes['V1405 Cas'] = -10.2 #we all wish!
filters[3] = 'SI'
```

Adding to lists and dictionaries

There are lots of ways to add data to a list:

```
# The built-in .append() method adds one data point to the end of
the list.
names.append('bet Per')

# You can also use the + symbol to append a whole extra list's
worth of data.
# This is known as 'concatenation'. (Concatenation works to combine
strings, too!)
names = names + ['WZ Cas', 'S CrB', 'SS Cyg']

# The .insert() method can add data to a list at any location, not
just the end.
names.insert(3, 'CH Cyg')
```

Here is how you can add a new piece of data (a “key:value pair”) to a dictionary:

```
peak_magnitudes['U Sco'] = 7.5
```

Indexing

Indexing is when you use square brackets `[]` to access the values inside of a collection.

Python starts counting at 0, so the first item in a list can be accessed by writing `[0]`, the second item by writing `[1]`, and so on. Starting at 0 might seem annoying at first, but it actually makes a lot of math easier once you start writing complex code.

Here is an example of using indexing to get values from a list:

```
filters[0] #Gets the first filter
filters[3] = 'SI' #Changes the fourth filter to 'SI'
```

You can use negative numbers to count backwards from the end of the list:

```
filters[-1] #Gets the last filter
```

Dictionaries can be indexed using their “key” values.

```
target['RA'] #Gets the right ascension of the target
peak_magnitudes['RS Oph'] #Gets the peak magnitude of RS Oph
```

True, False, and None

You can also use the words `True`, `False`, and `None` as values for variables. This can be useful for keeping track of things that don’t naturally make sense as numbers.

```
# Sometimes you might want to create a variable, even though you
don't actually have any data for it yet.
# None is perfect for those situations:
tonights_data = None

# True and False are great for keeping track of states and
statuses.
dome_open = False
taking_darks = True

# When you run your code, Python will turn each comparison into
either True or False. For example:
pixel_size > 10 #What you wrote
False #What Python sees
```

“Truthy” and “Falsey”

“Truthy” is the word for values which turn into `True` inside an if statement. “Falsey” is the word for values which turn into `False` inside an if statement.

In Python, all of these values are falsey:

```
falsey_keywords = [False, None] #Both False and None are falsey.
falsey_collections = [[],{},(),set()] #Empty collections are
falsey.
falsey_number = 0 #The number 0 is falsey.
falsey_string = '' #Empty strings are falsey. (They don't count as
empty if they have spaces in them, though.)
```

Every single other value is truthy.

Operators

Operators are used to do the most basic things in Python. They can be used to do math, check whether two values are the same, and more.

Here are examples of some useful operators:

```
# Simple math
x + y #Addition
x - y #Subtraction
x * y #Multiplication
x / y #Division

# More math
x ** y #Exponent: Raise x to the power of y.
x % y  #Modulo: Divide x by y, and return the remainder.
x // y #Floor division: Divide x by y, and round the result down to
the nearest whole number.

# Simple comparisons
x == y #Do x and y have the same value?
x != y #Do x and y have different values?

# More comparisons
x is y #Are x and y the exact same object? (This is useful because
you can give a variable multiple names.)
x in y #Is there something equal to x inside of y? (this assumes y
is a collection, like a list)

# Assignment (used to create and update variables)
x = 6  #x is now equal to 6
x += y #x is now equal to its old value + the value of y
```

You can view a full list of Python operators here:
https://www.w3schools.com/python/python_operators.asp

Careful: Pay attention to exponents

Sometimes, when writing about exponents, people use a caret mark `^` to mean "to the power of".

In Python, the caret mark `^` means "XOR", not "to the power of". (XOR is an obscure operator used for comparing binary numbers.)

If you use `^` in your exponents, your results will be wrong. Instead, you should use two asterisks `**` (like 'double multiplication'):

```
x ** y # "a to the power of b"
```

Functions

Functions are small pieces of code which do something specific. You can use a function just by typing its name, followed by parentheses `()`:

```
data = [4.5, 1000.2, 0.03, 1900]
max(data) #Gets the largest value inside data
```

Python comes with lots of built-in functions, and packages can provide even more. You can also write your own functions:

```python
def apply_dark(image, dark_frame): #Like a dictionary definition:
"def"ine a name for the function
    darked_image = image - dark_frame
    return darked_image

# Now we can use the function:
calibrated_frame = apply_dark(raw_frame, dark_frame)
```

Arguments

Functions need to be specifically given data to work with. Data passed into a function is known as “arguments” or “args”.

```python
min_jd = 2451545
max_jd = 2455198
bin_size = 30

# When you pass data into a function without using keywords=,
you're using "positional arguments".
# The function knows that if it gets a bunch of arguments without
labels, it should assume they are in the default order.
range(min_jd, max_jd, bin_size)

# When you label your input data using the keywords provided by the
function, you're using "keyword arguments", or "kwargs".
# Notice how using the keyword labels lets you type the arguments
in any order.
range(stop=max_jd, step=bin_size, start=min_jd)
```

Return

When a function is done running, it can either just stop, or it can provide some final data using the `return` keyword:

```python
def get_image(target):
    image = query(target)
    return image

def save_image(image):
    image.save()

image = get_image('Aldebaran') #Returns an image
saved_image = save_image(image) #saved_image is now equal to None,
because while save_image() did some stuff, it didn't return
anything.
```

Objects

An object is a collection of related variables and functions. In Python, **everything** is an object. Even the number `158023.2` is an object!

When a variable belongs to an object, it's called an "attribute". Similarly, when a function belongs to an object, it's called a "method". Even though the names are very different, attributes and methods are almost exactly the same as regular variables and functions.

Methods

Methods are functions which were defined inside an object. When you call them, you type the name of their parent object first, followed by a period `.`, and finally the name of the method.

```
do_thing() #regular function
thing.do() #method
```

Attributes

Attributes are variables which are attached to an object. You can access their values by typing the name of their parent object, followed by a period `.`, and finally the name of the attribute.

```
star.name = 'WZ Cas'
print(star.name)
```

Conditionals (if...elif...else)

Sometimes, you might want to run a piece of a code on some data, but only if a certain condition is met. You can do that using an "if statement".

```
if frame.exposure > 10:
    # This code will only run if the frame's exposure is greater
than 10.
    science_images.append(frame)
```

If you want to check for multiple conditions, you can add "elif:" statements after the if block:

```
if period > 200:
    print('This star is probably a Mira!')
elif period > 1:
    print('This star has a period between 1 and 200 days... it
could be anything!')
elif period < 1:
    print('This star is probably a short period pulsator!')
```

The "else:" keyword lets you add some code which will only run if the condition is **not** met:

```
if frame.exposure > 10:
    science_images.append(frame)
else:
    print("Warning: this image has an exposure shorter than 10
seconds, so it won't be included.")
```

Loops

Loops are one of the most powerful features of Python. Loops let you run parts of your code over and over again, in a process known as “iteration”.

The most common type of loop is the “for loop”. For loops take an “iterable” (an object with several other objects inside of it, like a list), and run your code on each of the items inside that iterable, one at a time. Here’s an example:

```
for frame in images:
    frame.save()
```

Loops run the same code every iteration, but that doesn’t mean they have to do the exact same thing every time–you can make loops that change the values of variables in real time. Here is an example of a loop which does something different every iteration:

```
for frame in images:
    if frame.exposure < 10:
        print('This exposure is too short! It must be a pointing
image.')
        continue # Skip ahead to the next iteration instead of
running the rest of the code in the loop

    frame = apply_dark(frame)
    science_images.append(frame)
```

Most of the time, there’s no need to know how many times a loop has run, but sometimes, you might find it useful to keep track. Python has a useful function for that called enumerate():

```
for i, frame in enumerate(images):
    if frame.exposure < 10:
        print(f'Frame number {i} is too short! It must a pointing
image.')
        continue

    frame = apply_dark(frame)
    science_images.append(frame)
```

Sometimes, you might not have an iterable to run the loop on, and you just want to run it a certain number of times instead. You can do that using the built in `range()` function:

```
for num in range(50):
    # This code will run 50 times
    print(num)
```

List comprehension

Another useful type of loop is the list comprehension. List comprehensions are like for loops, but shorter, and they automatically result in a list.

```python
# Both of these examples do exactly the same thing:
# "Create a list named exposures, and fill it up with the
.header['EXPTIME'] values from each image."

# For loop method
exposures = []
for frame in images:
    exposure = frame.header['EXPTIME']
    exposures.append(exposure)

# List comprehension method
exposures = [frame.header['EXPTIME'] for frame in images]
```

List comprehensions are best at simple tasks, not complicated ones. If you want to apply several steps of complex logic while building your list, you should stick to using a regular for loop–it'll be a lot easier to read. (Remember: shorter doesn't mean clearer.)

Packages (a.k.a. modules)

Packages are collections of code written by other people. They're one of the most powerful features of Python, allowing you to skip right ahead to what makes your program *new*, instead of having to waste effort on coding features which other people have already figured out.

Some of the most useful packages for astronomers include:

- Astropy – Contains tools for reading and writing many kinds of files (including FITS), handling light curves, spectra, and other astronomical datasets, and so much more.
- Astroquery – Related to Astropy. Allows fast & simple querys of services like Simbad, VizieR, and MAST, to help you do data mining.
- numpy – Contains useful math tools, as well as its own system for handling datasets in a very fast way.
- matplotlib – Allows you to very easily create all kinds of beautiful graphs and figures.

With the packages listed above, programs which used to require thousands of lines of code to write can be reduced to a single line. It's no wonder that almost every single astronomy script ever written includes those packages!

How to install packages

Before you can use a package for the very first time, you will need to install it. You can do that using pip, Python's built-in package installer.

Here's how:

1. Open your computer's terminal window. (On Windows, the terminal is known as the Command Prompt. On Mac, it's just called Terminal.)
2. Run this command: `pip install packageNameHere` (replace "packageNameHere" with the name of the package you want to install)

Importing

For each script which you create, if you want to use the features inside a package, you will need to import the package. You can do that using the `import` keyword. For example:

```
import astropy
```

From that line down, the astropy package will be available for use. (Since packages are only avaialble below the point where they were imported, it's customary to put all of your imports at the very top of your script file.)

Most of the time, you'll probably only want one or two of the features included in a package. You can import specific features like this:

```
from astropy.table import Table
```

To learn what kind of features are available in a package, and how to import them correctly, consult the package's official documentation.

Advanced features

This section covers Python features which are sometimes useful, but aren't necessary to write good code. They're generally rarer than the ones listed above, so you shouldn't worry too much about learning them when you're a beginner – focus on the basics first, and it will be quick and easy to learn these features if you ever need them.

Classes

Classes are blueprints for objects. Just like you can write your own functions using the keyword `def`, you can write your own objects using the keyword `class`:

```
class Star:
    # Method definitions go here
```

To be useful, every class should have a method called `__init__()`. The `__init__()` method is the default which will be run automatically whenever a new instance of your object is made.

```
class Star:
    def __init__(self, bmag, vmag):
        self.bmag = bmag
        self.vmag = vmag
```

Whenever a method is run, Python automatically passes a variable called `self` into the method, as the first argument. `self` contains all of the information about an object, so you can use it to access your object's attributes and run other methods.

Since Python always automatically adds `self` to the arguments, if you ***don't*** include `self` in the definition of your method, you'll get an exception:

```
class Galaxy:
    def __init__(redshift): #Notice how "self" was not included in
the definition!
        self.z = redshift

ic1101 = Galaxy(0.078)
```

```
---------------------------------------------------------------------
TypeError                           Traceback (most recent call last)
c:\Python dictionary.ipynb Cell 64 in <module>
              2     def __init__(magnitude): #Notice how "self" was
not included in the definition!
              3         self.magnitude = magnitude
        ----> 5         vega = Star(0)

TypeError: __init__() takes 1 positional argument but 2 were given
```

Instances

Instances are the objects you make using a class.

If a class is like a set of blueprints for a house, an instance is like an actual house. An instance can be changed, and store different data, but it will always be an instance of its parent class (just like how you can paint a house a different color, but it will always still be a house).

You can create a new instance by calling the class, as if it were a function (technically, you are calling the class' `__init__` function):

```
sirius = Star(-1.46, -1.46)
```

Each object has only one class, but there can be many instances:

```
vega = Star(0, 0)
arcturus = Star(1.18, -0.05)
deneb = Star(1.34, 1.25)
```

While loops

For loops aren't the only type of loop in Python. You can also create a loop using the `while` keyword.

While loops will run over and over again, as long as the value next to the `while` keyword is truthy.

```
while solar_altitude < -18:
    current_time = datetime.now()
    # This function "get_solar_altitude()" might return a number
greater than -18,
    # which would make the condition at the top of the while loop
no longer truthy, ending the loop.
    solar_altitude = get_solar_altitude(current_time)
```

You can make while loops which run forever:

```
while True:
    print('Oh no! An infinite loop!')
```

...but that's usually not a good idea, since none of the rest of your code can run while the while loop is running. Running an infinite loop might even freeze your computer.

How to troubleshoot Python code

Most of the time you spend writing code will probably be spent debugging. That's normal! Your first few bugs might be frustrating to solve, because you're still learning how to debug, but don't get discouraged–debugging gets easier with practice, and can even be quite fun (like solving puzzles).

Python has some special features which make it easier to debug. Here are two of the most valuable:

Exceptions

In Python, errors are called "exceptions". Here is an example of an exception:

```
my_list = ['has','three','items']
my_list[5] #Trying to access the 6th item (which doesn't exist!)
will raise an exception
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
c:\How to debug Python code.ipynb Cell 2 in <module>
      1      my_list = ['has','three','items']
----> 2      my_list[5]

IndexError: list index out of range
```

Exceptions always come with error messages to help you figure out what went wrong. In the example above, the error message says "list index out of range". That tells us that the error was caused when the Python interpreter tried to index (look inside) a list, but there was no item inside the list at the given index.

i: Exceptions are normal

90% of the time, when you test out your code, it will crash with an exception. That's just how debugging works; it doesn't mean you write bad code!

Raising an exception

A piece of code can "raise an exception" if it detects something wrong. When an exception is raised, it causes the code which is currently running to crash. This is often done on purpose, to help make it easier to troubleshoot bugs. (The error message will probably be more useful to you if your code crashes at the exact point where it first detects a bug, instead of waiting until the very end to try to figure out what went wrong.)

You can raise your own exceptions, if you want. This can be useful if you know you want your program to fail under a certain set of conditions. For example, you might want to raise an exception if the FITS file you just opened is completely empty:

```
if len(fits_table) == 0:
    raise ValueError('The input FITS file was empty. Please try
again with a FITS file containing data.')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
c:\How to debug Python code.ipynb Cell 4 in <module>
      1     fits_table = []
      3     if len(fits_table) == 0:
----> 4         raise ValueError('The input FITS file was empty.
Please try again with a FITS file containing data.')

ValueError: The input FITS file was empty. Please try again with a
FITS file containing data.
```

Tracebacks

A traceback is displayed whenever code crashes due to an exception. It shows the lines of code which were being run when the exception occurred.

Here is an example of a traceback:

```
Traceback (most recent call last):
  File "C:\Users\Lauren\Desktop\process_spectra.py", line 1, in <module>
    from astropy.table import Table
ModuleNotFoundError: No module named 'astropy'
```

This traceback says that it tried to import the Table object from Astropy like we asked, but it turned out that Astropy isn't installed.

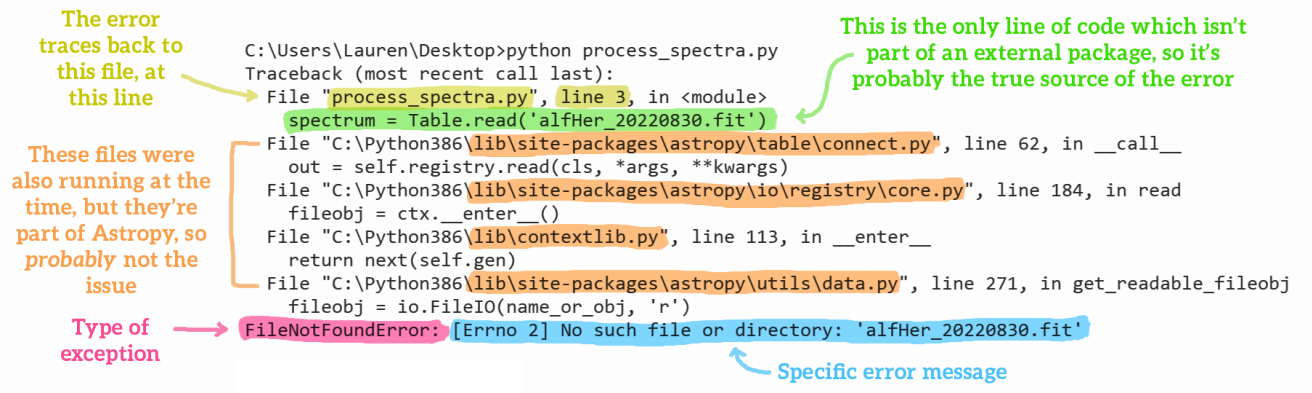
Parts of a traceback

The traceback always starts with `Traceback (most recent call last):`, so when you're scanning the output of your code looking for errors, keep an eye out for that line.

The main body of the traceback is made up of a list of the lines of code which were running at the exact moment the exception was raised. Each line of code is tagged with the name of the file where it can be found. In general, you can ignore lines which are in files that weren't written by you.

The last part of a traceback shows the type of exception which was raised, followed by the error message attached to this specific exception. (This error message often contains the specific value which had just been given to the troublesome line of code, triggering the exception.)

Example: Reading a traceback



Tracebacks might seem hard to understand at first, but they are ***very*** useful! 99% of the time, a traceback contains all of the information you need to debug your code.

That said, not even experts know what every traceback means on first glance. The information is all there, but it's often written in such a way that you need to have a little background knowledge about the code being run. For that reason...

Don't be afraid to Google!

If you're having trouble with a particular bit of code, first, try to understand the traceback. If you get stuck, don't sit there and struggle; copy-paste the traceback into your search bar! (Well, okay, not the *whole* traceback: usually the search results are most accurate if you only copy-paste the last line.)

If you find a solution that you like, adapt it for your code, and use it. Remember, this isn't school. Your work isn't being graded on originality. In fact, it's usually a *good* thing if your code is unoriginal! Hundreds of thousands of very smart people have been working on perfecting coding for years, and they've got it down to a science... why not learn from them?

Where to go to learn more

There is a lot of useful information about Python out there on the internet, but there is also a lot of confusingly-written junk. To get you started, here are a few websites which are generally good resources for learning:

- Stack Overflow - A site where you can ask questions about programming. Answers to past questions are archived, and can show up in Google searches, making this one of the most useful resources when debugging exceptions.
- W3schools - A site full of well-written, interactive Python tutorials. Focuses on the fundamental features of the language, but has good information on other topics, too.

- Geeks4Geeks - A site full of Python tutorials, usually decently written. Covers some more advanced topics than W3schools.

By Lauren Herrington