# Python Best Practices

By definition, “best practices” are generally agreed upon procedures that are accepted as being most effective---innately, these are going to be context specific. Some best practices are more universal than others. For example, it is generally agreed that clear documentation is a “best practice” regardless of the programming language you are writing in. Others, like following PEP8 style, are going to be very language specific (PEP8 is specifically for Python---PEP stands for “Python Enhancement Proposal”). The following list of best practices range from the general to the specific and assume that the coding you’re doing is in Python with the audience being other scientists.

## Jacob’s Best Practices

1. Follow the PEP8 style-guide
2. Write modularly (related: use the `main` function, unless file is only a library)
3. Thoroughly and consistently comment and document your code (also see here)
4. User `argparse` and `configparser` for input handling
5. Handle exceptions, set exit codes, and log your errors
6. Write informative GitHub README (also, to see an **excellent** template see here)

## #1 PEP8 style-guide

Well formatted code is a requirement of this class and a good practice to maintain in all of your projects. Code is read much more often than it is written, and it is in your interest to make that code as readable as possible. Comments are also important, but they are not a replacement for good code. Each coding language has its own standards for what constitutes “good code” within that language. So that there is no ambiguity about what counts as well-formatted code, we will be using PEP8, which is the official formatting standard for Python.

There are many specifics included in PEP8. The following are just a few examples of how to format your code, however keep in mind that all the “not recommended” versions are still valid code, in that they will still run, but they are considered to be less *human readable*. Note, in some cases there are acceptable alternatives:

- **Variable names:** variable names should be informative characters, words, or words and be formatted in “snake case” (all lowercase characters with underscores between parts). Avoid ‘l’, ‘o’, and ‘I’ since they can be interpreted as ‘0’ or ‘1’

```
# Recommended
x, var, my_variable

# Not recommended
l, VARIABLE, myresultvariable
```

- **Function names:** functions should be an informative word or words in snake case.

```
# Recommended
func, compute_function

# Not recommended
f, FUNCTION, myComputeFunction
```

- **Class names:** Classes should be an informative word or words with the first letters capitalized and without underscores to separate them.

```
# Recommended
Model, MyClass

# Not recommended
model, my_class
```

- **Whitespace in expressions:** No whitespace around “=” in function arguments. A single space around lowest priority operations in mathematical expressions. Single space around “=” in variable assignment or any other assignment operator or comparison operator.

```
# Recommended
def function(default_parameter=5):
    # ...

# Not recommended
def function(default_parameter = 5):
    # …


# Recommended
y = x**2 + 5
z = (x+y) * (x-y)

# Not recommended
y = x ** 2 + 5
z = (x + y) * (x - y)


# Recommended
My_variable = 5

# Not recommended
my_variable=5
```

- **Indentation:** Indentation is used to separate code blocks. Each instance of indentation should be exactly 4 space characters (NOT tabs).

```
# Recommended
x = 3
if x>0:
    print(x)
else:
    print("Not positive")


# Not recommended
x = 3
if x>1:
 print(x)
else:
      print("Not positive")
```

- **Maximum line length**: individual lines should be limited to 79 characters
- **Line wrapping**: For lines longer than 79 characters line continuation is done by wrapping the line in brackets/parentheses and matching the indentation ("implied continuation")

```
# Recommended
total = [first_variable, second_variable, third_variable,
         fourth_variable, fifth_variable]


# Recommended alt ('hanging indent')
total = [
     first_variable, second_variable, third_variable,
    fourth_variable, fifth_variable
]


# Not recommended
total = [first_variable, second_variable, third_variable,
    fourth_variable, fifth_variable]
```

## Using pycodestyle

There are many many many more stylistic nuances to PEP8 (many seemingly arbitrary ones) and it can be a lot to keep track of. However, there are packages that can help you to correctly format your code and teach you the PEP8 style as you code. The most widely used is `pycodestyle`. This package installs a command line tool that you run on your python code file. It will output any instance where your code does not meet PEP8 standards and state the applicable style code.

The following is an example of a python script `bad_practice.py` I wrote that has a number of non-PEP8-compliant issues. The general workflow is to run this tool frequently as you write

code (say before you make each commit), fix any issues as they come up, and ultimately your script shouldn't return any non-PEP8-compliant lines.

```
$ pycodestyle bad_practice.py
bad_practice.py:2:10: E401 multiple imports on one line
bad_practice.py:3:1: E302 expected 2 blank lines, found 0
bad_practice.py:3:8: E211 whitespace before '('
bad_practice.py:3:10: E201 whitespace after '('
bad_practice.py:3:12: E203 whitespace before ','
bad_practice.py:3:16: E202 whitespace before ')'
bad_practice.py:4:3: E111 indentation is not a multiple of four
bad_practice.py:4:4: E225 missing whitespace around operator
bad_practice.py:5:3: E111 indentation is not a multiple of four
bad_practice.py:7:1: E302 expected 2 blank lines, found 1
bad_practice.py:7:10: E201 whitespace after '('
bad_practice.py:8:4: E111 indentation is not a multiple of four
bad_practice.py:8:12: E225 missing whitespace around operator
bad_practice.py:9:4: E111 indentation is not a multiple of four
bad_practice.py:11:1: E305 expected 2 blank lines after class or function
definition, found 1
bad_practice.py:11:80: E501 line too long (90 > 79 characters)
bad_practice.py:12:3: E128 continuation line under-indented for visual indent
$
```

A secondary benefit to using pycodestyle is that it will teach you PEP8 in the process of using it. As you use this tool while writing more and more Python code, PEP8 style will become second nature and you'll get to a point where you rarely have to fix any style issues.

Two helpful options for pycodestyle that you might want to use while learning how to code in Python are `--show-source`, which will explicitly state the line of your code that has the error and point to its location, and `--show-pep8` which will state the PEP8 specifications for each error and will show good and bad examples for each one.

## #2 Modular coding

Modular coding is an approach that emphasizes segmenting the functionality of an entire program into independent, discrete components, each one having all the necessary information to execute one aspect of the whole programs' functionality. Each segment is then written as a single function in your code.

This is best described by example. Consider the following (silly) example of a script, in which I want to generate a list of '`n`' of random integers between '`a`' and '`b`' and then calculate the mean and standard deviation of these, and finally print them to STDOUT.

In the linear approach below, you see everything is processed in sequence, with the input parameters being specified at the very start. In the modular approach, you see that the three different steps of the process (1. generating random numbers, 2. calculating their average, and 3. calculating their standard deviation) are each contained in a reusable, general purpose function definition and only at the end are they called with the specific input parameters. See the following comparison:

Linear approach:

```
import random

n = 100
a = 10
b = 70
rand_list = []

for _ in range(n):
    rand_val = random.randint(a, b)
    rand_list.append(rand_val)

rand_avg = sum(rand_list)/len(rand_list)

rand_diff_sqr = []
for i in rand_list:
    rand_diff_sqr.append((i - rand_avg)**2)

rand_stdev = (sum(rand_diff_sqr)/len(rand_diff_sqr))**0.5

print("Avg: ", rand_avg)
print("Stdev: ", rand_stdev)
```

Modular approach:

```
import random

def rand_generator(a, b, n):
    rand_list = []
    for _ in range(n):
        rand_val = random.randint(a, b)
        rand_list.append(rand_val)
    return rand_list

def calc_avg(values):
    avg = sum(values)/len(values)
    return avg

def calc_stdev(values):
    avg = sum(values)/len(values)
    diff_sqr = []
    for i in values:
        diff_sqr.append((i - avg)**2)
    stdev = (sum(diff_sqr)/len(diff_sqr))**0.5
    return stdev

values = rand_generator(10, 70, 100)
print("Avg: ", calc_avg(values))
print("Stdev: ", calc_stdev(values))
```

The benefit of the modular approach is that the three functions can be reused in other contexts whereas the code in the linear approach is only applicable to this single script. Furthermore, the

modular approach can be read and understood in predefined “chunks,” making it easier for someone else to make sense of your code.

Generally speaking, when writing your code, try to think “could I put this small bit of code into a function definition?” Doing so will result in a modular piece of reusable code that is ultimately going to be more readable as well.

## __name__ and __main__

Modularity is a useful practice in the context of python for more than just the reasons outlined above. One of the other key reasons is related to some of the special case variables in Python, which are internal to the language (these are variables that have double-underscores, or “dunders”, at the start and end---see this link).

In particular, we are concerned with the `__name__` variable. A feature of a Python file is that there is no explicit signal for whether this file is meant to be executed at the command line as a script OR imported as a library into another file. This is where the `__name__` variable comes in. This variable stores the name of the file/script containing it. However, that “name” is different depending on how you *use* the file. Consider the following example. Say I create the following file, which just contains a print statement of the variable:

File: `name_var.py`

```
print(“Name variable: ”, __name__)
```

Now let’s look at what happens if that file is run at the command line OR if it is imported.

Imported:

```
$ python
>>> import name_var
Name variable: name_var
>>>
```

Command line:

```
$ python name_var.py
Name variable: __main__
$
```

The `__name__` variable takes on different values, depending on how you use the file! Specifically, if you ever execute the file at the command line, it takes on the value `‘__main__’`.

So now you might be asking “Ok, so what? How is this helpful? What does this have to do with modularity?” Well, this brings us to the canonical architecture of a python file that enables it to be both imported AND executed.

Architecture of a pythonic `.py` file:

```
# Modular functions defined first
def func1( … ):
    …
    return …

def func2( … ):
    …
    return …

def func3( … ):
    …
    return …

...

# main function def which performs all steps using modular functions above
def main( … )
    …
    return …

# Run main if file is being executed as a script
if __name__ == '__main__':
    main( … )
```

Here we see if we import the file, nothing is executed, but all the functions can be used in the import context. Conversely, if this file is executed all execution is carried out by the `main()` function.

Beyond the above canonical example, there are other ways of using the __name__ variable. For example, if you want to make a file only intended for import or only intended for execution. The following two examples show how those files might be structured (NOTE: this is not a very common practice, unlike the previous example, but can be useful in some contexts)

In both cases, the `raise` method is used within a logical check of the `__name__` variable. The `raise` method is a way of triggering a specific error which is then passed to STDERR and halts execution of the program. By running this logical check at the start of the file you immediately terminate the code if it is being run in the wrong context.

Note that in the example that's only intended for execution, we still define a `main()` function. By naming the function this way you indicate to other individuals reading your code that it contains the principal functionality of your executable file. So, while there is no hard rule requiring you to use this naming convention, it is a good one to follow for future readability.

A `.py` file only for import:

```
# Prevent the file from being executed
if __name__ == '__main__':
    raise RuntimeError("This module is intended only for import.")
```

```
def func1( … ):
    …
    return …

def func2( … ):
    …
    return …

...
```

A `.py` file only for execution:

```
# Prevent the file from being executed
if __name__ != '__main__':
    raise RuntimeError("This module is not intended for import.")

def func1( … ):
    …
    return …

def func2( … ):
    …
    return …

...

# main function def which performs all steps using modular functions above
def main( … )
    …
    return …

main( … )
```

In the above file architectures all the code is contained in function definitions (i.e. it is all modular), the same as the first canonical example, therefore no code is executed unless one of those functions is called. This allows for the code to be imported or executed without side effects.

## #4 Commenting and Documenting

Commenting and documenting (two different things) are ways of explaining to a user or developer what your code is supposed to do. “Comments” are small statements embedded in your code that explain nuances or details of your code that may be difficult for a developer to understand simply by looking at the code itself. **The audience for comments are you and other developers.** Comments do not impact the way the code runs. Comments in python are indicated by the octothorpe character (‘`#`’)---all characters following the comment character in a line will be ignored by the Python interpreter. These comments can help to explain the reasoning behind the associated code. For example:

```
# this comment could describe what this file is
```

```
file = open(filename, 'r')

# This comment could describe what this for-loop is doing
for line in file:

    values = line.strip().split(',')

    # This comment could explain the purpose of the if-statement
    if some_check:
        do_something(values)
    else:
        do_something_else(values)

…
```

The comments are helpful because they can be granular and closely associated to the code they intend to describe. However, it is good practice to keep your comments brief! If you are writing comments that are multiple lines long, you might want to consider making your code more straightforward, so it doesn't need so much explanation.

Documenting is different from commenting, and in Python your documenting is done through what they call "`docstrings`". Unlike comments, **the audience for docstrings are the users of your software**.

Docstrings are what are returned when you run the `help()` command on a package or function. Take for example if I look at the help for a function in the numpy package:

```
>>> import numpy
>>> help(numpy.asarray)

Help on built-in function asarray in module numpy:

asarray(...)
    asarray(a, dtype=None, order=None, *, like=None)

    Convert the input to an array.

    Parameters
    ----------
    a : array_like
        Input data, in any form that can be converted to an array.  This
        includes lists, lists of tuples, tuples, tuples of tuples, tuples
        of lists and ndarrays.
    dtype : data-type, optional
        By default, the data-type is inferred from the input data.
    order : {'C', 'F', 'A', 'K'}, optional
        Memory layout.  'A' and 'K' depend on the order of input array a.
        'C' row-major (C-style),
        'F' column-major (Fortran-style) memory representation.
        'A' (any) means 'F' if `a` is Fortran contiguous, 'C' otherwise
        'K' (keep) preserve input order
        Defaults to 'C'.
    like : array_like
```

```
        Reference object to allow the creation of arrays which are not
        NumPy arrays. If an array-like passed in as ``like`` supports
        the ``__array_function__`` protocol, the result will be defined
        by it. In this case, it ensures the creation of an array object
        compatible with that passed in via this argument.

        .. versionadded:: 1.20.0

    Returns
    -------
    out : ndarray
        Array interpretation of `a`.  No copy is performed if the input
        is already an ndarray with matching dtype and order.  If `a` is a
        subclass of ndarray, a base class ndarray is returned.
```

Here we see a thorough explanation of the input parameters to this function as well as what it returns. This gives the user insight into how the function should be utilized. There are a number of formats used for docstrings, across all Python packages, however **we will be using the same format numpy uses for its docstrings for all the documenting of our code.**

There are two locations for docstrings: 1) at the top of your library files and 2) within each function definition. The former is printed if `help()` is run on the package itself and the latter is printed if `help()` is run on the specific function (as is the case in the above `numpy.asarray` example).

So how does one create a docstring? Easy. All all docstring is is a special type of string that is distinguished by three quotations or apostrophes (e.g., `"""This is a docstring"""` and `'''This is also a docstring'''`). So what does this look like in your Python library file? Let's look at the following partial code example.

File `my_lib.py`:

```
"""
This is the main docstring for my library file. It contains high level
information about the package which describes its overarching purpose.
"""'

def my_func(a, b, c=None):
    """
    This function does some things that are helpful.

    Parameters
    ----------
    a : int
        This parameter is the first one. It does the following things…

    b : float
        This second parameter work with the first like this…

    c : string
```

```
        This parameter is the only keyword argument. It is a string that
        does something different than either `a` or `b`. It's default value
        is `None`


    Returns
    -------
    my_list : list
        The function return is a list that has information in it which was
        affected by the choice of values for our input parameters
    """

    …
    # Below is all the actual code for the function
    …
```

Here we see the two different types of docstrings. The docstring associated with the function definition has a lot of detail, and the format you see here is the same as that for numpy. There are two sections "Parameters" which defines the input parameters (specifying the required data type and an explanation of what the parameter is used for) and a "Returns" section which defines what the outputs of the functions are (specifying the returned data type and an explanation of its contents).**Every single function in your library should have a docstring formatted in this fashion!**

Once you've made docstrings for all the functions in your library you can check them by running the `help()` function at the Python command line on the imported package.

## #5 Argparse and Configparser

The two Python packages `argparse` and `configparser` are useful tools for structuring the inputs to your software. Proper formatting of your software inputs makes the software easier for the user to understand how to run and makes the results more reproducible.

### Argparse

The `argparse` package formats your command line inputs and standardizes how you specify the parameters. The `configparser` package allows for the creation of a "config file" which will contain all the input parameter information, which your software will read from for all the input values.

The general structure for using argparse in your script is as follows: 1) import the package, 2) create an `ArgumentParser` object from the package, 3) add arguments to the parser object with `add_argument()`, and finally 4) run the `parse_args()` routine on your parser object. What will be returned is a dictionary containing the formatted input arguments. See the below skeleton code:

```
import argparse

parser = argparse.ArgumentParser(description="My useful program.")
```

```
parser.add_argument(  <options for an argument>   )
parser.add_argument(   <options for another one>   )
…

my_args_dict = parser.parse_args()
```

Say you want your piece of software to have two inputs: 1) a required string (say a filename to be opened) and 2) a pair of integers. These specifics would be included in the options of their respective `.add_argument()` command. For example:

```
# input argument 'a' is our required string
parser.add_argument(
    '-a', '--mystr',
    type=str,
    action='store',
    required=True,
    help="This string here is required!")

# input argument 'b' is our pair of integers
parser.add_argument(
    '-b', '--bvals',
    type=int,
    nargs=2,
    action='store',
    required=True,
    help="This is a pair of integers")
```

Here we see that the first parameters supplied to `add_arguments()` are the ways you call the arguments at the command line (exactly like flags for a command line tool in bash). We then see we specify a data type for the argument (`"type"`) and the number of values that are included (`"nargs"` --- the default is 1). We then specify that the value specified is to be saved in the final parsed args dictionary (`"action"`). Next we specify whether or not the argument is required. Lastly, we include a help message that can be printed out at the command line to help the user understand the purpose of the argument.

These are just some of the options you can use to specify the format for your input arguments. To see the full documentation on these and many more features, go to this link.

One of the major benefits of using argparse is that it will automate the creation of a manual for your piece of software. This manual helps users understand how to run your software. Once the above has been included into your code, the manual can be printed just like that for any other command line tool, using the `-h` flag. See the following for the two arguments above.

```
$ python my_argparser.py -h
usage: my_argparser.py [-h] -a MYSTR [-b BVALS BVALS]

My useful program.

optional arguments:
  -h, --help            show this help message and exit
```

```
  -a MYSTR, --mystr MYSTR
                        This string here is required!
  -b BVALS BVALS, --bvals BVALS BVALS
                        This is a pair of integers
$
```

Here you can see the automatically generated manual specifies the usage information (with optional arguments indicated by square brackets) as well as your high level description, and then lists the various arguments along with their flags and the help info.

## Configparser

The `configparser` package can be used in conjunction with `argparse` to provide the inputs for your software from a file (a "config" file). This improves your reproducibility because the inputs to your software must be saved and this file can then be provided to anyone attempting to run your software. There are several config file formats but the configparser package uses the INI file format. This consists of headers (defined by square brackets) that delineate sections. Within each section each input parameter is specified in the format `key=value` (alternatively, you can also use ':' as a delimiter, as in `key : value`). See the following for an example:

File `config_file.ini`:

```
[sec1_name]
param1 = 10
param2 = 'value'
…

[sec2_name]
variable1 = [1, 2, 3]
variable2 = True

...
```

While only one section header is necessary, they can help to keep your input parameters organized in the event that you have many parameters/arguments for your software. It is recommended that you group them thematically. For example, say you have a number of files that are involved in your software (both as inputs and outputs), you may then want to create a `[FILES]` section header in your config file. Additionally, say you have many parameters that control how your results are formatted and processed, then maybe you would want a `[RESULTS]` section header under which you can include those. The format for the name of these sections can be any string, however the parameters you specify must obey Python variable naming conventions.

In order to input this file, you simply need to import the configparser library, create a `ConfigParser()` object and run the `.read()` submethod. See the following example that demonstrates how to input the config file and reference the parameters once parsed.
File `config_read.py`:

```
import configparser

config = configparser.ConfigParser()
config.read("./config_file.ini")

# Reference a variable from the first section
config['sec1_name']['param1']

# Reference a variable from the second section
config['sec2_name']['variable2']
```

## #4 Handle exceptions, set exit codes, and log errors

It is inevitable that you will experience an error in your code. An "error" is when the code's execution is prematurely halted due to the computer's inability to interpret or process some portion of the commands or data that have been provided by the user. There are many many ways of triggering errors, and all programming languages have some way of addressing them.

In Python, the word used for error is an "exception." When an exception is raised (that is, an error occurred when executing the code), the Python interpreter will return information detailing what line in the code caused the exception and the type of exception that occurred, often with a helpful message. This is what is called the "traceback" (sometimes referred to as a "stack trace"). The goal of this best practice is to address specific exceptions one anticipates or observes in one's code, and to provide informative error messages for the user that are specific to the operation of the software. In this way, one can guide the user to use the code more efficiently and effectively.

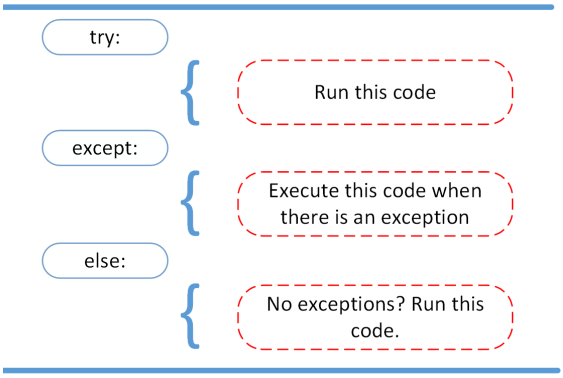
For a list of all the built-in exceptions see this link. There are two main approaches to "handling" exceptions:

1. Catch the exception when it occurs using the `try/except` construct.
2. Proactively raise an exception if a necessary condition is not met using the `if/raise` construct

These two methods are described in the following sections.

### Try/except

The main tool Python has for dealing with exceptions is another control flow construction, akin to `if/else`, called "`try/except`". In its most basic form, `try` and `except` each demarcate a code block (much like `if/else`). First, the `try` block is run. If no exception is raised in this block then the `except` block of code is skipped. However, if an exception DOES occur then the `try` block is terminated and the corresponding `except` block is run instead. An



`except` block can be specified for each individual type of exception. Lastly, you can also include an `else` block, which will run in the event that the `try` block is completed successfully (see the figure above).

**NOTE:** *The `try/except` construction is designed so that in the event of an error, your code can continue running! After the `except` code block runs, the remainder of the code continues.*

The general approach is if you have a piece of code which you know has the potential for an error or you have observed to have an error, this code should be placed into a `try` block. The corresponding `except` block (or blocks) should then be written such that all anticipated or observed errors that could occur in the `try` block are addressed.

There are many different built-in exceptions (i.e., different types of errors)---for example, `ImportError` when a module can't be loaded, `IndexError` when an index of an iterable is incorrect, `NameError` when a variable name is not found, `TypeError` when an operation is applied to an object of incorrect type, etc---which are called based on the code context. In general, you should try to call each `except` clause specifying a specific exception. For example, say you want to open a file, but want to catch the case where the file may not exist. In this case you would use the FileNotFoundError exception as follows:

```
import sys
file_name = "./my_file.txt"

try:
    file_handle = open(file_name, 'r')
except FileNotFoundError:
    print("FileNotFoundError: could not open the file.", file=sys.stderr)

...
```

Assuming the file does not exist, when the above is run we get the following result returned to STDERR:

```
FileNotFoundError: could not open the file.
```

In this example, we are anticipating the possibility that the file doesn't exist, so we are defining an `except` clause that specifically addresses it. Here you can see we are using the STDERR pipe (imported from the `sys` module---i.e. `sys.stderr`) as our destination for the error message. This allows us to separate this message from the STDOUT, and redirect the message into an error file in our bash script in which we run the python code.

Often, in order to incorporate exception handling for specific exceptions, you will need to discover the types of exceptions your code is actually subject to (by running your code under a range of conditions). The more experience you get coding in Python the better you will be at anticipating what exceptions are likely to occur, which will save you time in the testing process.

There are a number of ways of utilizing the `try/except` construction, however there is one way you should definitely NOT use it.

How ***NOT*** to do exception handling:

```
try:
    do_stuff()
except:
    pass
```

**THIS IS A BARE, SILENT EXCEPTION AND SHOULD ALWAYS BE AVOIDED.** It is "bare" because the `except` clause has no specific exception stated, so it catches all possible ones (any possible error that can occur in `do_stuff()`). It is "silent" because the `` `pass` `` statement literally does nothing (it is the "null statement" in Python) and then the program will continue to run without any record of the error occurring.

However, a bare exception is not always a bad option. If you intend to use a bare exception, it is generally good practice to record in your error log file the ENTIRETY of the traceback produced by the code. This way, even if the error message may be excessive and confusing, you will at least have a record of it. Say we want to rewrite the above "file opening" example, but with a bare `except` clause. In order to access the traceback, we can import one of Python's base modules, aptly named `traceback`. See the following:

```
import traceback
file_name = "./my_file.txt"

try:
    file_handle = open(file_name, 'r')
except:
    print(traceback.format_exc(), file=sys.stderr)

...
```

Assuming the file "`./my_file.txt`" does not exist, when the above is run we get the following result returned to STDERR:

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'my_file.txt'
```

Indeed, we see the exception being produced is the `FileNotFoundError` we had anticipated. NOTE: we used `sys.stderr` as destination for our print message (set by the "`file=`" kwarg), which will allow us to redirect the error message using "`2>`" in our `run.sh` script. The default destination for the print statement is STDOUT.

For more detail see this exception overview, this discussion of bare exceptions, and this overview of tracebacks.

## if/raise

The alternative (or compliment) to *catching* an exception and redirecting your code around it, is to first check if a necessary condition is met before an exception occurs, and if the condition is not met then actively raising an exception. Checking the condition is done with an if-statement, and within the if-statement code block you can use the `raise` function to actively raise your chosen exception. ***Once an exception is raised, the code is aborted.***

**NOTE**: There is both a major philosophical and procedure difference here, relative to `try/except`. The former method (`try/except`) allows you to cause an error but continue with the rest of your code (the error occurs in the `try` block so instead the `except` block is executed and the rest of the code continues afterward). Conversely, if/raise allows you to *anticipate* an error and *actively* raise an exception that will stop your code. Try to think about circumstances where one behavior would be preferable over the other.

The syntax of the raise function is as follows, where `ExceptionName` is one of the official Python exceptions or a user created one:

```
raise ExceptionName("My error message for the exception.")
```

For example, say a particular variable `my_var` is supposed to be an integer. If it is of any other data type, you would want a `TypeError` exception to be raised, which can be done as follows:

File `raise_example.py`:

```
my_var = '5'

if not isinstance(my_var, int):
    raise TypeError("variable `my_var` is not of type int.")
```

Here we are using the `isinstance()` function which checks if the first argument is of the type specified in the second argument and returns a boolean. We are then applying the `` `not` `` operator to it to negate that value. Since `my_var` has been defined as a string, `isinstance` returns `False`, which is negated to `True` and the if-statement is satisfied.

Command line:

```
$ python raise_example.py
Traceback (most recent call last):
  File "raise_example.py", line 2, in <module>
TypeError: variable `my_var` is not of type int.
$
```

Furthermore, say `my_var` is not only supposed to be an integer but also an integer that's greater than 10. In the event it's not greater than 10, you would then want to raise a `ValueError`. This could be done as follows:

File `raise_example.py`:

```
my_var = 5

if isinstance(my_var, int):
    if my_var > 10:
        pass
    else:
        raise ValueError("`my_var` is an int but not greater than 10.")
else:
    raise TypeError("variable `my_var` is not of type int.")
```

Here we are accounting for both possible undesirable conditions and raising the appropriate exception so that we can distinguish between the two.

```
$ python raise_example.py
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: `my_var` is an int but not greater than 10.
$
```

In the event that `my_var` is an int that's greater than 10, the `pass` statement would be executed, no exception would be raised, and the program would continue running.

Thus, in this way, the `if/raise` construction can be used to catch increasingly subtle or complex errors, by nesting or stringing together `if/elif/else` statements.

Regardless of which of the two methods you use, the more you can anticipate and actively address exceptions/errors in your code and provide informative error messages, the more robust and user friendly your code will be.

## Exit codes and logging errors

All software (at least that which runs on a Unix machine) generates what is called an "exit code." Simply, this is an integer value that indicates whether a piece of software ran successfully and if not what the possible error might have been. When a program runs "successfully" (that is, it was not terminated prematurely due to an error) it returns the exit code "0". In the event it doesn't run successfully it returns a non-zero integer (simplest case is an exit code "1" as a catchall for any error). The exit code is stored in a bash variable "`$?`" which gets overwritten by each subsequent command. You can access the exit code of the previous command by echoing it. Look at the following examples:

```
$ pwd
/home/jovyan
$ echo $?
0
$ let "var=1/0"
bash: let: var1=1/0: division by 0 (error token is "0")
$ echo $?
1
$ ls /asdlkfhlkasjhdf/
```

```
ls: cannot access '/asdlkfhlkasjhdf/': No such file or directory
$ echo $?
2
$ notacommand
bash: notacommand: command not found
$ echo $?
127
```

Above, the first command “pwd” was the only one to generate a 0 exit code because it was the only valid command. All the rest produced some non-zero exit that clues what the potential error was. For more detail on the various error exit codes and what they indicate see here.

`Try/except` and `if/raise` are our two ways of addressing the errors *where they occur within Python*, but exit codes and logging the STDERR stream with “2>” is how we can record the errors outside the Python environment. Let’s revisit the file opening example from the start of the section. It is possible that you will want the program to be interrupted in the event that the file does not open. In this case you should (in the except block) print an error message to STDERR, trigger a system exit, and set the exit code, as follows.

Part of `my_script.py`:

```
import sys
import traceback
file_name = sys.argv[1]

try:
    file_handle = open(file_name, 'r')
except:
    print(traceback.format_exc(), file=sys.stderr)
    exit_code = 1
    sys.exit(exit_code)


...
```

Now the code will no longer continue to run after the except block is complete, since sys.exit() will terminate the script. We can then write our `run.sh` script to document the error.

File run.sh:

```
#!/bin/bash

input="./my_file.txt"
output="./my_results.txt"
err="./error_log.txt"

python my_script.py $input > $output 2> $err

echo Exit code: $?
```

Now when we run it we are alerted to our error at the command line (via the exit code) and the details of the error are logged to a specific file for reference.

```
$ run.sh
Exit code: 1
$ cat ./error_log.txt
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'my_file.txt'
$
```

Ultimately, this is a far more robust way of tracking errors in our code during development and for the benefit of the users of your software.

## #6 Write informative READMEs

The last of the best practices is to write an informative README as a part of your GitHub repo. This README is the first piece of documentation that will be seen by anyone using your software. For this reason, it should contain all the information they will need to successfully use your software. Some examples of the type of information to include: installation instructions, user tutorial, contact information, unit testing results, changes log, etc. Instead of going into detail here, I leave it up to the reader to take a look at the two links provided at the top of the page. And, to get started on your own repo README, I suggest downloading this template.