Python Chases Monkey

An Introduction to Python Programming

Jason M. Pittman

Chapter 6

Python Looks Inside

Introduction

It is time for Phil the python to gain a measure of introspection. Perhaps counting its scales has led the python to question why it has scales in the first place. Alternatively, maybe Phil has seen the monkey and is pondering from where the instinct to chase it originates. Of course, the same can be said for the monkey.

In all honesty, this is a fancy way of indicating that we have enough going on in our programs to consider *auxillary* topics. Some of you might be wondering why we're covering these topics in the middle of the book as opposed to the end. If I'm being honest, it isn't because exception handling, efficiency, debugging, or logging are more important than a topic such as **data structures**. Instead, these topics are important enough that we ought to begin incorporating them into our programming lexicon sooner rather than later. As well, our *python chases monkey* program is becomming large enough to warrant introducing these ideas now.

With that, let's step back and take in the entire scene. The tree canopy is full and green. The monkey has frozen to avoid being seen by the python. The python is uncoiled, gently tasting the air to locate the source of the movement it has heard. Both animals begin to consider their options…

Learning Outcomes

1. Distinguish between types of exceptions and the means to handle each within a program
2. Utilize the standard python debugger to resolve errors and test code
3. Log internal program information to external endpoints

Exception Handling

A brief definition is in order before we start exploring **exceptions** in detail. **Exceptions** are error conditions that arise during runtime in otherwise *correct* code. We can const

Broadly speaking, **exceptions** can be separated into two forms: built-in and user-defined. Both operate identically albeit one is implicit (built-in) and the other must be explicitly called (user-defined). Further, the important takeaway now is that we must comes to grips with how to handle both forms of exceptions.

Built-in

Let's do a quick Python experiment. Imagine the monkey wants to calculate the number of bananas available across all of the trees within its immediate area. Being a monkey, it doesn't use the calculator correctly. To simulate what a monkey might do, run `print(0 / 0)` in your interpreter.

We should expect to get an error since computers don't handle division by zero. Specifically in this case, we ought to see `ZeroDivisionError: division by zero` returned by the interpreter. Perfect.

This little experiment demonstrates one of the many built-in forms of exceptions. However, we knew ahead of time that we would get an exception. What about handling exceptions that may arise unexpectedly? Python is rather smart in this regard and includes two keywords for us to use: `try` and `except`. Rather than belabor the point, let's take our earlier experiment and translate it into a new example with these keywords.

Example 1

```
def divide():
    try:
        print(0 / 0)
    except Exception as e:
        print(e)
```

There are two points of interest in **Example 1**. First, `try` and `except` form a pairwise pattern for **exception handling**. That is, `try` sets up an attempt to execute a statement while `except` gives us an opportunity to do something if the result of the attempted statement is an exception. Then, secondly, we can execute any logic within the scope of the `except` block. That's a pretty cool trick to know about so that we don't become trapped (he he he) into thinking we can only output the exception.

Moreover, we can stack our `except` blocks as shown in **Example 2**.

Example 2

```
def divide(x, y):
    try:
        print(x / y)
    except ZeroDivision as e_zero:
        print(e_zero)
    except Exception as e:
        print(e)
```

This example shows how we can stack the handling of exceptions so that we can respond differently to different conditions. This is a more advanced technique but can be quite useful when we have code capable of producing varying levels of exceptions. One tip: always place the more specific exception catch before less specific blocks in these cases.

Lastly, if we are interested in producing robust code, and we should be, then wrapping our code in `try-except` blocks is one of the strongest habits to develop. This applies to built-in functionality as well as the functionality we implement in our own types.

User-defined

It stands to reason that since we are designing our own **types** (e.g. `Snake`, `Animal`, `Monkey` and so forth), we can also define our own **exceptions**. This makes further sense when we realize the implicit operation in the built-in form of an **exception** includes a `raise` call. More technically, we use `raise` to *throw an exception.* Consider the following:

Example 3

```
def Count(snake):
    if snake.scales > 9000:
        raise Exception("A snake cannot have over 9000 scales...")
```

Here we use `raise` to throw a custom exception when the number of scales a given snake has exceeds a specified value. What we see in **Example 3** is a rough prototype of the type of processing that occurs within a built-in **exception** such as `ZeroDivision`. This means we still have to wrap the `Count()` call in a `try-except` block elsewhere to hanlde whatever exception we define. Speaking of which, we're not limited to general `Exception` either. We can use any built-in exception such as `TypeError` or `NameError`. The type of exception is less important right now than the habit of coding with the idea of validating results and throwing the exception using `raise`.

Of course, an **exception** tells us when there is a problem during runtime. What if we just want to know how well our program is running?

Efficiency

The notion of efficiency is not explored often in an introductory programming course. Yet, I suggest knowing just how well or how *optimally* are code executes is critical to understanding programming. Efficiency is also how we might differentiate between two seemingly equivalent solutions.

Efficiency is measurable. I would go so far as to say that efficiency is *easily* measurable. Wait, I think we should go one step further and agree that all code should include an efficiency measuring harness as part of its debugging structures. Hold up, actually, this is a great time to introduce the formal idea of **unit testing** which is something we informally practice as we iteratively run our programs as we add statements to see if any exceptions are introduced.

Unit testing

For illustrative purposes, let's say we need to caclulate the volume of a plant so that our python and monkey can *see* them in 3D space. We might have code like the following:

Example 4a

```
def calculate_volume(l, w, h):
    return (l * w * h)
```

Looks good right?

Well, if we run: `calculate_volume(2, 0.4, 'ten')` we will receive a `TypeError` exception. We have to resist the urge to exclaim, *well, who would pass a non-int type to this function?!?*. The responsibility is entirely ours to implement robust code. Thus, it falls to us to understand how and when our code might fail. That means **unit testing**.

Example 5

```
import unittest
from plant import Plant

class PlantTester(unittest.TestCase):
    p = Plant()

    def test_volume(self):
        result = p.calculate_volume(1, 2, 3)
        self.assertEqual(result, 6)

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

Two new Python programming constructs appear in **Example 5**. We have the module `unittest` and the class method `assertEqual()`. There are a host of methods we can use but for now we are just focusing on testing if the arithmetic result of the volume calculation given the specified input `(1, 2, 3)` matches the indicated unit test value `6`.

We also see something new in how we run our unit test class: `python3 plant_tester.py -v` insofar as we use the `-v` switch to produce *verbose* output. Doing so will produce shell output like:

```
test_volume (__main__.PlantTester) ... ok
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

This is a two-for-one. Not only did you validate the code operates as expected but we also generated an efficiency metric evidenced by the `Ran 1 test in 0.000s` output. There is a much deeper conversation to be had regarding efficiency but now we have a built-in Pythonic means to produce measures. Since we should be unit testing all of our work anyway, we might as well use the test run time value to compare and contrast implementations.

I should also point out somewhat strongly that it is a good idea to also test what should be failing outcomes. A failing test will produce an `AssertionError` with details of the failing outcome. Needless to say, we cannot test every possible input. Thus, we need the means to find out why exceptions are produced in code that otherwise has passed unit testing. That's why we have **debugging**.

Debugging

Speaking of debugging, I would rank general debugging skills as more desirable than advanced programming knowledge. There is a straightforward rationale for my assertion: if you can debug, you'll be able to fix bugs in someone else's code as well as your own. If you cannot debug effectively, you will struggle to troubleshoot just your own code. Simple.

Also *simple* is the first thing we can do when we run our program and encounter an unhandled exception. We run again but wit the `-i` parameter as in `python3 -i myprogram.py`. This will automatically load an interactive python interpreter session when the error occurs. Once in the interactive session, we can check what values are set in fields, what objects are instantiated, and so forth. The shortcoming though is that we cannot interact with the program as it is running. To do that, we need a true debugging option.

To start, there are three options we have available to run our code in *debugging* mode. First, if we have run with `-i` we can manually load the Python debugger, `pdb` in the resulting interactive interpreter session. This is done by importing the pdb module as `import pdb` and then running `pdb.pm()`. The `.pm()` method is a *post-mortem* or after-the-fact inspection. We'll leave *what* we can do within the `pdb` shell momentarily.

Next, we can also change the way we run our python program in a slightly different manner. Whereas up to this point we simply ran `python3 myprogram.py` or in an equivalent manner, now we need to add `-m pdb` to the command. Using the same idea, for debugging we'd run `python3 -m pdb myprogram.py`.

Example 6

```
def spawn_number(x):
    spawns = 1
    for i in range(1, x + 1):
        spawns = spawns * 1

    return spawns
```

Running **Example 6** using `python3 -m pdb` will load an interactive debugging shell. Then we can use commands such as `step` to step through each line, `n` to execute the *next* statement and `p [symbol]` to print the value of indicated symbolic name.

If we are going to get good with debugging, we also need to know how to set breakpoints in our code. For our convenience, Python has a `breakpoint()` function. We load back into the `pdb` interactive debugger and use `b file:line` to jump to the indicated execution point.

Pretty cool, right? Sometimes however employing `pdb` can be overkill. Maybe we just want a quick check of sorts, just to see if we're doing okay as we build the overall program.

Print

I would not be doing a good job of showing how most of us carry out our Python programming if I didn't talk about (ab)using the `print()` function.

Use the `print()` function has to be the most common form of debugging, at least what we can agree is *informal* debugging. Leveraging `print()` has the advantages of not requiring specialized knowledge and being easy to implement. In contrast, `print()` has the disadvantages of being reactive and working at a low level of resolution.

The best metaphor I can summon is that using `print()` as a debugger is like hearing someone talk on the phone. You passively hear one side of the conversation and then only in an after-the-fact manner.

The setup is simple enough. Take **Example 7** for instance:

Example 7

```
def calculate_volume(l, w, h):
    print(l * w * h)
    return (l * w * h)
```

This will show us the result *before* we return it. This would provide a view into the `calculate_volume()` function inner-workings which we can compare and contrast to the returned value after the fact.

Put simple, if you are going to employ `print()` for informal debugging, be sure to (a) `print()` before and after as well as (b) remove or disable the `print()` calls when you're done.

Logging

The motivation for logging is, in my view, the same for having a window; you want to be able to see. However, in the case of programming, the *seeing* occurs in the reverse direction. We want, we need to be able to see into the program. More specifically, we want to see the state of the program; the values, program flow, and results of computational processing. Further, we want to do so even when there are no exceptions, no errors. Yes, exactly- logging is not just for problems but also for tracking the behavior of our programs even when it behaves well.

I don't want to get too far into a technical implementation because we deal with files in a later chapter. For now, let's stick with the concept of logging and dig into *why* and *how* to log while leaving *where* for a later discussion. Doing so will serve as a stronger bridge between our debugging and file I/O conversations.

We have two implementation techniques when it comes to implementing logging. First, we can design our own logging mechanism. This can be as simple as writing to a file instead of using `print()`. Second, we can use the `logging` module included in the base Python language architecture. The difference is material and certainly the labor involved is even more different. However, no one can reasonably tell you one way is better than the other. There is a rule of thumb though.

My advice: use the native `logging` module unless you have a specific reason not to. Using an existing wheel is almost always better than reinventing it. Logging is definitely a wheel that is not worth redoing on our own unless we have a very specific reason to do so.

Something to know about the default `logging` module is that we can specify different levels of logging. Here, we can connect the concept of logging to the practice of debugging and create a bridge of sorts between the reactive `print()` method and the active `pdb` technique.

Exercises

1. Modify **Example 1** so that the function executes a valid division operation in the `except` block.
2. Incorporate a `try-except` block in the `length` setter class method you designed based on **Example 5** in chapter 2.
3. Add a `raise` to the `Animal` class constructor such that if a `str` type is not provided as `name` we throw an `Exception`.

Questions

1. What is the difference, if any, between an **exception** and a **syntax error**?
2. Why do we position more specific `except` blocks before general blocks?
3. What specifically causes the `TypeError` in **Example 4** and how would you fix it?