ICS 32 Winter 2014
Code Example: Exceptional Control Flow

Background

Understanding the difference between success and failure

When a *function* is *called* in Python, that function is being asked to do some kind of job. The function does the job and *returns* a result — always an object of some type, though that object might be the special object **None** if the function's role is to generate some kind of side effect (such as printing output) rather than calculating and giving you back a result. As you've seen, many functions accept *parameters*, which allow the function to do a slightly different job each time it's called; for example, a function that downloads an image from the Internet would probably take at least one parameter, the address from which the image should be downloaded, so that the function could potentially by used to download any image instead of just a particular image.

The interaction between a function and its caller in Python has at least some similarity to certain kinds of interactions between people. Think about what happens you ask a friend to do something for you, like "Here's $5. Can you drive over to Starbucks and buy me a latte?", which, conceptually, is a lot like calling a function in Python (and "$5", "Starbucks", and "latte" are its parameters). Even assuming your friend understands your instructions perfectly and is willing to do it, are you guaranteed to get the result you asked for — in this case, a latte? — or are there circumstances where you won't get it? It doesn't take much thought to realize that failure is a possibility here. Your friend's car might not be in working order (or it might be in use by someone else, or your friend might not even have one!). Starbucks might be closed, or they might have run out of coffee. A latte might cost more than $5.

Now let's think again about a Python function that downloads an image from the Internet. Even assuming that the function is perfectly written, can anything go wrong there? Sure! Your Internet connection might not be working. The web site from which you're trying to download the image might be down, it might not contain the image you asked for, or it might not exist at all. What result should the function return in this case? Going back to the previous example, when you send someone to Starbucks and it turns out that Starbucks is closed, you get no result at all; instead of handing you a coffee, your friend might instead inform you that the job couldn't be done and *why*. "Sorry," your friend might say, "I couldn't get that coffee for you, because Starbucks was closed." Or, right away, your friend might say, "Are you crazy? I don't have a car, remember?!" Either way, you're not getting the coffee you wanted.

In Python, when a function is called, it is being asked to do a job. Broadly speaking, just like in the case of sending your friend for coffee, there are two possible outcomes, even assuming the function has no bugs:

- The function will complete its job successfully and return an object of a type you expect.
- The function will fail to complete its job. Functions fail differently than they succeed in Python; rather than just returning an object that indicates failure, they don't return an object at all, but instead *raise an exception*.

Despite their name, there's nothing exceptional about exceptions. They're not rare, they're not necessarily indicative of bugs, and they don't cause well-written programs to crash. An exception just means a function failed to complete its job. Where some finesse is required is in deciding what should be done about it.

What happens when an exception is raised

An exception that is not handled anywhere in a program will cause a crash and you'll see a *traceback*, which specifies information about the unhandled exception and where the program was in its execution at the time the exception was raised. For example, consider this nonsensical Python module.

oops.py

```
def f():
    x = 3
    g(x)

def g(n):
    print(len(n))

if __name__ == '__main__':
    f()
```

If you run this module in IDLE, you'll see this result, which offers some insight about what happens when an exception is called in Python.

```
Traceback (most recent call last):
  File "C:\Example\oops.py", line 11, in
    f()
  File "C:\Example\oops.py", line 3, in f
    g(x)
  File "C:\Example\oops.py", line 7, in g
    print(len(n))
TypeError: object of type 'int' has no len()
```

Reading a traceback from the bottom up provides a lot of useful information.

- The type of the exception — exceptions are objects in Python, just like everything else — is a **TypeError**.
- A more descriptive account of the problem is **object of type 'int' has no len()**. That's a hint that we were trying to get the length of an integer; integers have no length.
- The exception was actually raised on line 7 of **oops.py**, by code in the function **g()**. (The traceback even tells us the code that's on line 7.)
- The function **g()** had been called by the function **f()**, on line 3 of **oops.py**.
- The function **f()** had been called by the "main" **if** statement, on line 11 of **oops.py**.

Given all that information, it doesn't take long to figure out what happened:

- **f()** was called.
- **f()** initialized a local variable **x** to the integer value 3. So **x** has the type **int**.
- **f()** called **g()** and passed **x** to **g()**'s parameter **n**. So, within **g()**, **n** also had type **int**.
- **g()** attempted to get the length of **n**. But **n** was an **int** and **int**s have no length! An exception was raised.

Any exception raised within a function that does not contain any code that handles exceptions will cause that function to fail. Control proceeds to whatever function called that function, which has the same option: either handle it or step aside. And so on. If no one handles the exception, eventually all functions fail; at that point, you'll see a traceback. In this case, **g()** raised the exception and didn't handle it; **f()** didn't handle it, either; and the "main" **if** statement didn't handle it, either. So we saw a traceback.

Catching an exception

We specify what should happen in a function when exceptions are raised by writing a **try** statement. A **try** statement is built out of *clauses* and is structured like this:

```
try:
    statements that will be attempted once
    if any exception is raised, control leaves the "try" clause immediately
except:
    statements that will execute after any statement in the "try" clause raises an exception
else:
    statements that will execute after leaving the "try", but only if no exception was raised
finally:
    statements that will always execute after leaving the "try", whether an exception was raised or not
```

There are two combinations of these clauses that are legal; other combinations are illegal because they are nonsensical. (Think about why.) In both cases, the clauses must be listed in the order below:

- A **try** and a **finally** and nothing else
- A **try**, at least one **except**, (optionally) an **else**, and (optionally) a **finally**

except clauses can optionally — and, more often than not, they do — specify a type of exception that they handle. Python only executes **except** clauses when the type of exception matches the type that the **except** clause can handle. **except** clauses with no type listed can handle any kind of exception.

Handling exceptions appropriately

Once you understand the mechanics of how a construct in Python behaves, your next task is understanding the appropriate ways to use it; no part of a programming language is right for every circumstance. We've now seen how you can handle exceptions, but the more nuanced problem is understand *when* to handle them and *when not* to handle them. Here are a few guidelines to consider.

- When you're writing a function **f** that calls another function **g**, one thing you want to be thinking about is whether **g** raises exceptions (i.e., whether **g** can fail to complete its job). If so, you then need to consider whether the failure of **g** also implies the failure of **f**, or whether **f** could reasonably carry on in some way. If **f** fails whenever **g** fails, you won't want to catch the exception in **f**; if **f** could reasonably continue, **f** should catch the exception and then continue its work.
- All in all, in my experience, it's more common for a function *not* to catch an exception than it is to catch it. This is especially true when you tend to write relatively short, simple functions, and break larger, more complex functions into smaller ones, as you should be doing, because it's more likely that the failure of one will cause a cascading failure of several others that are

each doing a smaller slice of something bigger.
- The **finally** clause is primarily used for what you might call *cleanup*. This is most especially true when a function acquires some kind of external resource — like a file or a connection across a network — that only it (and the functions it calls) will use. In the case of a file, for example, a **finally** clause provides an obvious place to close the file if was was opened.
- When you don't want to catch an exception, but you do want to ensure that cleanup is done when an exception is raised, a **try**/**finally** (with no **except** or **else**) is appropriate. That way, if the code in the **try** statement completes successfully *or* if an exception is raised and the function is fails, the cleanup in the **finally** will always be done.

The code

With that in mind, here is the code example from the previous lecture, as well as an alternative example that we didn't have a chance to cover. One example shows a scenario where you might consider catching an exception where it's raised, because there's a reasonable way to handle it and still have the function complete its work; another example shows a scenario where catching an exception where it's raised is clearly wrong, because there's no way for the function to complete its work without dramatically contorting its purpose.

Each example is presented in a single commented Python module, linked below.

- The **print_lines** module
- The **line_count** module