Errors

Errors happen all the time. Sometimes you misspell something or you forget a character or you use the wrong variable. Regardless of what happened, Python is very unhappy with the code it is trying to execute and tells you instead that the code cannot be run because of a certain reason. Othertimes, Python will execute the code happily, but the result is completely unexpected and not what is desired. Although you don't get a formal error message, this can be considered an error. In ENGR 102, these errors are put into three categories:

- Syntax errors
- Runtime errors
- Logic errors

A list of common errors I see students run into in ENGR 102 can be found in common_errors.ipynb, which should be in the same folder as this set of notes.

Syntax Errors

Syntax errors are arguably the simplest errors to deal with. These errors arise when Python checks your code first before executing it. Common syntax errors include missing commas and misspelling important names (such as variables, functions, and modules). In other words, syntax errors are pretty much typos or grammar errors: they can be caught immediately without needing further context since they violate basic Python rules. Example 10-1 shows a sample code that contains a syntax error:

In [ ]:
```
# Exmaple 10-1: Syntax error

print("Hi)
```

```
  Cell In[1], line 3
    print("Hi)
          ^
SyntaxError: unterminated string literal (detected at line 3)
```

As you can see, Python does not execute the code but just reports the error. Neatly enough, the type of error is indeed a `SyntaxError` but more specifically, it's an "unterminated string literal" error. That's just the formal way of saying that I need to close the string with a ".

Runtime Errors

Runtime errors happen when the code looks correct but a fatal occurs during execution (hence the error type name: the error occured during runtime). Python is pretty bad at guessing and so it can be quite bad at predicting when things can go wrong. In ENGR 102, this can be problematic since Python is very, very flexible (compared to other coding languages). The most seen runtime error, at least in my opinion, is trying to do something to data that is the wrong type (such as trying to add a string with an integer):

In [ ]:
```
# Example 10-2: Runtime error

print("5" + 5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[2], line 3
      1 # Example 2: Runtime error
----> 3 print("5" + 5)

TypeError: can only concatenate str (not "int") to str
```

Visually, Python is fine with the code: the `print` statement looks to be written out correctly, the addition operation has two things to add, and the string is closed properly. However, note that Python was unable to spot that I was adding a string and an integer. IDEs may be smart enough to warn me against doing such a thing, but looking strictly at *syntax*, Python assumes all is well and then executes the code. It is when Python tries to add `"5"` and `5` together is when Python realizes that the operation is illegal and it throws an error.

Logic Errors

Logic errors are different than the other two types because of the fact that logic errors do not produce actual errors that Python can throw. The error in this case is not with the code but with the *logic*. That is, the code is written technically correctly, but it does not produce the expected results. These errors are the hardest to locate since the issue could happen anywhere in the code, and the more complex the code, the more things that can possibly go wrong. A super basic logic error is shown with Example 10-3:

In [ ]:
```
# Example 10-3: Logic error

# Goal: Count all even numbers between 1 to 10
count = 0
for i in range(1, 10):
    if i % 2 == 0:
        count += 1
```

```
print(count)
```

```
4
```

This may *seem* right, but there are actually five even numbers (2, 4, 6, 8, 10). The logic error here stems from accidentally excluding out 10 as `for` loops do not include the upper bound. The simple fix would be to change the 10 to an 11.

In [ ]:
```
# Example 10-4: Logic error fixed

count = 0
for i in range(1, 11):
    if i % 2 == 0:
        count += 1

print(count)
```

```
5
```

You may have noticed that the module eight analysis notes actually show ways to help debug and prevent these errors from happening (although syntax errors are generally just a quick scan and fix).

Exceptions

Formally, errors are considered to be exceptional cirumstances. Hence code that helps prevent these exceptional circumstances fall under the umbrella of exception-handling. As the name suggests, this is code that helps Python continue executing the code even when an exception happens, allowing the code to continue instead of being halted. The error from Example 10-2 is an example of an exception.

Try/Except

The methodology we use to catch exceptions is to try some code and to execute it except when an error happens. Neatly enough, this is implemented in what is called a "try-except" block. In other languages, you may see this term as "try-catch" (as in you try the code and then catch the error before the language gets angry at you). The format of this is shown in Example 10-5.

In [ ]:
```
# Example 10-5: Basic try/except

try:
    x = int(input("Enter a number: "))
    print(x * 2)
except:
    print("That's not a number!")
```

```
That's not a number!
```

In Example 5, Python will first run the code in the `try` block. If any exception occurs, it will immediately stop running the code it is currently on, and Python will start executin the code in the `except` block. When I ran Example 5, my input was "tt". Trying to convert "tt to an int (due to the `int()` cast I have) caused an exception. But in this case, instead of stopping the program and giving an error statement, Python went to the `except` block and executed the `print` statement instead.

Note that if all the code in the `try` block executes correctly, then all the code in the `except` block is skipped since the `except` block is only executed when the code in the `try` block above it runs into an execption. A very common usage of `try/except` is for input validation:

In [ ]:
```
# Example 10-6: Input validation

# Get initial input
x = input("Enter a number: ")

# Infinite loops are generally bad, but it is exceptionally awkward to not use an infinite loop in this case
while True:
    try:
        print("Value attempted to convert:", x)
        x = int(x)
        print("Converted to int!")
        break

    except:
        print("Please enter a number!")
        x = input("Enter a number: ")
```

```
Value attempted to convert: 9.8
Please enter a number!
Value attempted to convert: tt
Please enter a number!
Value attempted to convert: 100
Converted to int!
```

This is my personal template code for verifying that an input is an `int` (change `int()` to `float()` to verify that the input is a `float`). Note that you should be using `try` for code that may break. Asking for input does not cause errors, so I leave that outside. It would be fine if I left it inside, however, this would mean that I may run into issues using custom `input` prompts when exceptions are handled. The code I present is arguably the most flexible for any sort of changes you need. Feel free to change to play around as needed

and to run the code in a normal .py file.

General Debugging Skills

I covered the basics of debugging in module eight, particulary the three main ways to debug:

- Unit testing
- `print` statements
- Debugger

Unit testing is also a way of bug *prevention*: testing your code using values to ensure that it should works under normal circumstances and that the code doesn't get messed up with weird inputs. But once you get that logic error, it's probably best to test that input and see what happens and check if the unexpected behavior can be replicated for similar inputs.

Using `print` statements is a great way of tracking variables and code states to isolate certain execution states where the problems start to rise. If you have a runtime or logic error, you can output the offending variable or output values leading up to the offending block of code to see where stuff goes wrong. Once the weird behavior has been found out, you now have a good starting point for figuring out how to fix your code.

A debugger is great for detailed step-by-step analysis. You can see every single variable and their values at any line of execution. When unit testing and `print` statements are not in-depth enough or it's too difficult or complicated to use the other two, use the debugger.