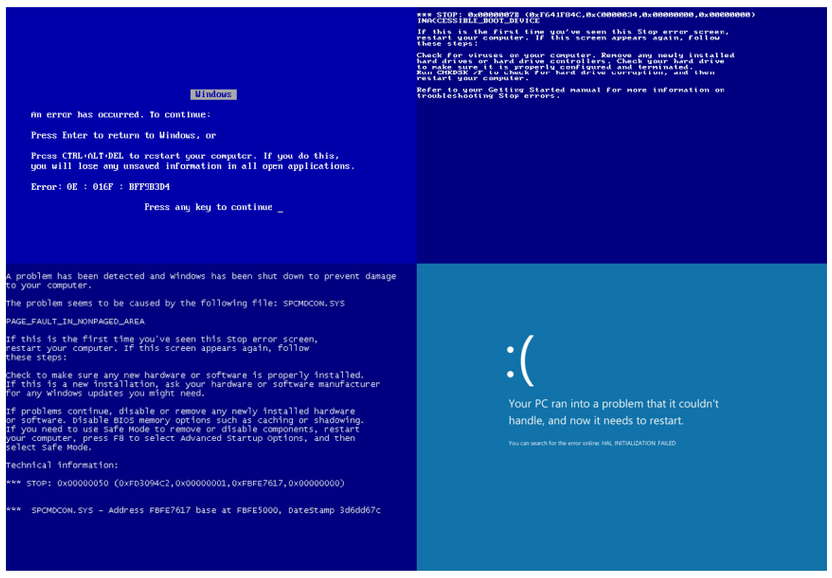
Week 2 - Exceptions

Updated Sept 15, 4:10pm

One of the implicit lessons of last week's lab was in what we didn't specify: what happens when the user makes mistakes? Last week's exercise explored built-in Python **runtime errors**, which occur *while a Python program is running* when severe errors occur, like trying to access a variable that hasn't been declared, or dividing by 0. Any one of these halts the execution of the program. But if we're trying to write robust code for software, we don't want our programs to crash!



Taken from http://www.techiesguide.com/evolution-of-blue-screen-of-death.html"

In this lecture, we'll explore two aspects of a common error handling strategy known as **exceptions**: how to create and raise them, and how to deal with them gracefully.

Two common strategies

It's common in some languages for functions to return special values indicating an error has occurred; for example, one might think to return `None` in Python, and in Unix several functions return `-1` to indicate an erroneous result. There are two issues with this strategy:

- Values reserved for errors might be confused with valid return values
- Whenever the function is called, an extra check must be made in the code to see if the return value represents an error

In practice, this strategy might result in a variable silently getting assigned a null value because of an error, only to cause the whole system to blow up dozens or hundreds of steps (lines of code, function calls, etc.) later!

Another common practice is colloquially called "Look Before You Leap" (LYBL): if you know that a function dos not work correctly for certain inputs, then before calling the function, check whether the inputs are valid. While this is certainly a good approach, and the most common in many other languages like Java, this isn't the approach Python uses, for a few different reasons:

- Python is more permissive than other languages (like Java) when it comes to assigning variable values, and this makes it harder to check for every erroneous value that might occur
- Depending on the complexity of your system, it might be the case that something

changes between the check and the actual function call, rendering the check pointless
- Even when you're confident your values won't change, it is often very difficult to predict all of the possible erroneous *values* that will cause an error (but easier to predict that actual errors themselves). This is especially true when dealing with external libraries.

What are exceptions, and how can you use them?

In contrast to LBYL, Python takes the approach "It's Easier to Ask Forgiveness Than Permission" (EAFP). Conceptually, it is a simple flip of LBYL: call the function on the (possibly erroneous) inputs, and then deal with any errors that might arise.

The rest of this note is about two central topics: raising errors to signify a problem, and then deailing with them gracefully so that our programs do not crash.

In Python, **exceptions** are just regular objects that represent an exceptional situation (not necessarily a problem). The runtime errors you explored last week are built-in exceptions, but you can easily define your own depending on the design of your program.

For example, consider the situation of creating a Person class specifying a name and age. We could do this pretty simply as follows:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

There are many ways this could go wrong if we're given invalid inputs. A simple one is that the age can't be a negative number! We can create our own **exception** class that represents this situation, and then **raise** it in the constructor.

```
class NegativeAgeError(Exception):
    pass

class Person:
    def __init__(self, name, age):
        if age < 0:
            raise NegativeAgeError

        self.name = name
        self.age = age
```

The creation of the `NegativeAgeError` class looks different that our regular classes; we'll talk much more about that for next class, but for now just remember that any exception class you define *must* have the `(Exception)` following the class name. And while more sophisticated exceptions can have their own attributes and methods, here we're more interested in simply using the object to represent the error.

So to us, the really important part is the `raise` keyword, used to (surprise!) raise an exception. The important thing to understand is that this signals an `NegativeAgeError` exception and **immediately terminates the method**, without allowing it to finish. Thus no `Person` object will be created with a negative age, and instead whenever the constructor is called with a negative age argument, an error is produced. Try calling

`Person('david', -10)` in the interpreter to see for yourself!

Handling Exceptions

Okay, so that's how we can create and raise exceptions. How do we deal with them? After all, doing nothing but raising exceptions just causes more runtime errors.

In the console:

```
>>> Person('Janus', -5)
Traceback ...
NegativeAgeError
```

In another function:

```
def make_neg_person():
    return Person('Janus', -5)

>>> make_neg_person()
Traceback...
NegativeAgeError
```

Even in a nested series of function calls:

```
def make_person(n):
    return Person('Janus', n)

def make_neg_person():
    return make_person(-5)

>>> make_neg_person():
Traceback...
NegativeAgeError
```

Suppose we have a function `f` that calls another function `g`, and `g` raises and exception. Control returns to `f`, which might contain some code to deal with the exception; if it has this code, we say that `f` **catches** or **handles** the exception, and the program can continue safely. Otherwise, `f` raises the same error, returns control to *its* calling function, and so on until either the exception is caught at some point, or no function can handle the exception and the program crashes.

Okay, so how do you catch an exception? The basic Python construct is called a `try-except` block:

```
def safe_neg_person(n):
    try:
        x = Person('Janus', n)
        print('Successfully created a Person!')
    except NegativeAgeError:
        print('You tried to create a negative-age Person. Shame on you!')

    print('Program continues safely, even if NegativeAgeError was raised.')
```

What's going on here? The call to the constructor is in a `try` block. If this raises a `NegativeAgeError` exception, the exception is caught by the `except NegativeAgeError`, and

the code inside this block gets executed. In either case, control flow proceeds normally after the block, and the final `print` statement is always executed.

We use `except` blocks to catch specific errors: this way we know exactly what we're dealing with. Python allows you to catch all errors by using `except Exception:`, but this should generally be avoided. Otherwise, we run the risk of ignoring *unexpected* errors, only to have them cause problems down the line.

```
def safe_neg_person(n):
    try:
        x = Person('Janus', n)
        print('Successfully created a Person!')
    except Exception:
        print('You tried to create a negative-age Person. Shame on you!')

    print('Program continues safely, even if NegativeAgeError was raised.')

safe_neg_person('not a number lawl')
```

(In class, we also saw simply `except: ...`, which is even more general because it captures things like Keyboard Interruptions with `Ctrl+Z`.)

To catch multiple exceptions from the same block, you can attach multiple `except` blocks to the initial `try` block.

```
def safe_neg_person(n):
    try:
        x = Person('Janus', n)
        print('Successfully created a Person!')
    except NegativeAgeError:
        print('You tried to create a negative-age Person. Shame on you!')
    except TypeError:
        print('You need to pass in a numeric argument.')

    print('Program continues safely, even if NegativeAgeError or TypeError was raised.')
```

Optional: Else and Finally blocks

Suppose we made a slight change to the previous example:

```
def safe_neg_person(n):
    try:
        x = Person('Janus', n)
        print('Successfully created a Person with age ' + n)
    except NegativeAgeError:
        print('You tried to create a negative-age Person. Shame on you!')
    except TypeError:
        print('You need to pass in a numeric argument.')

    print('Program continues safely, even if NegativeAgeError or TypeError was raised.')

>>> safe_neg_person(10)
You need to pass in a numeric argument.
Program continues safely, even if NegativeAgeError or TypeError was raised.
```

What happened? We made a small error (adding a string to an int), but this was

inadvertently caught by the second `except` block. The problem is that we only wanted to catch errors in the `Person` creation, only execute the print statement when this was successful, but not catch any errors that the print statement raises, because these are truly unexpected. We can combine the `try-except` block with an `else` block to accomplish exactly this:

```
def safe_neg_person(n):
    try:
        x = Person('Janus', n)
    except NegativeAgeError:
        print('You tried to create a negative-age Person. Shame on you!')
    except TypeError:
        print('You need to pass in a numeric argument.')
    else:
        print('Successfully created a Person with age ' + n)

    print('Program continues safely, even if NegativeAgeError or TypeError was raised.')
```

There might also be other errors we fail to predict, which would cause `safe_neg_person` to raise exceptions. However, it's possible that we would want some code to execute in every case, even if an unforseen error occurs. We can put such code in a `finally` block:

```
def safe_neg_person(n):
    try:
        x = Person('Janus', n)
    except NegativeAgeError:
        print('You tried to create a negative-age Person. Shame on you!')
    except TypeError:
        print('You need to pass in a numeric argument.')
    else:
        print('Successfully created a Person with age ' + n)
    finally:
        print('This always prints, even if an unexpected error occurs.')

    print('Program continues safely, even if NegativeAgeError or TypeError was raised.')
```

Try to see what happens when `safe_neg_person(-10)` and `safe_neg_person(10)` are called!

The `finally` block even executes when an error occurs in the `except` or `else` blocks - but notice that even if it runs, it *doesn't handle the error*. That is, if an unexpected error occurs anywhere else in the `try-except-else` block, the code in the `finally` block will execute, but it will not prevent the error for halting the execution of the function.

One common example of when this is useful is when reading from a file: if some unexpected error occurs, it is good practice to close the file in a `finally` block before passing control back to the calling function. This is basically what the `with` statement does for us when reading and writing files; without it, we'd be using a try-except block ourselves.

Optional: a more sophisticated Exception class

Since this came up a few times in lecture, let's see a simple example of adding a bit more to our custom exception class to improve the quality of the error reporting.

```
class NegativeAgeError(Exception):
    def __init__(self, age):
```

```
        """ (NegativeAgeError, int) -> NoneType
        Create a new NegativeAgeError with the given (bad) age.
        """
        self.age = age

    def __str__(self):
        """ (NegativeAgeError) -> str
        Return a string representation of the error.
        """
        return 'Tried to create a Person with negative age {}'.format(self.age)

class Person:
    def __init__(self, name, age):
        if age < 0:
            # Note the explicit parameter passed to the constructor now
            raise NegativeAgeError(age)
        else:
            self.name = name
            self.age = age
```

```
>>> Person('Janus', -10)
...
NegativeAgeError: Tried to create a Person with negative age -10
```

Notice that by defining the special method `__str__` in our error class, we can make a custom message print on the command line when the error is raised! Awesome.

By the way, what exactly do we mean by "special"? We'll talk about that much more next class, when we learn about inheritance!

David Liu (liudavid at cs dot toronto dot edu)
Come find me in BA4260
Site design by Deyu Wang (dyw999 at gmail dot com)