COMP 364: Errors and Packages

Carlos G. Oliver, Christopher Cameron

October 4, 2017
Outline

1. Recap + Warmup
2. Bugs
3. Useful packages
4. Practice Problem
Recap

- Anatomy of a function, namespaces
- Importing
Warm-up

- Write a program that asks the users for a list of names and grades and stores them in a list.
- Print the name of the person with the highest GPA.
Bugs: when things break

- You will probably have noticed by now that things don’t always go as expected when you try to run your code.
- We call this kind of occurrence a “bug”.
- One of the first uses of the term was in 1946 when Grace Hopper’s software wasn’t working due to an actual moth being stuck in her computer.

---

1Wikipedia
Types of bugs

There are three major ways your code can go wrong.

1. Syntax errors
2. Exceptions (runtime)
3. Logical errors
Syntax Errors: \text{“Furiously sleep ideas green colorless.”}\textsuperscript{2}

- When you get a syntax error it means you violated a writing rule and the interpreter doesn’t know how to run your code.
- Your program will crash without running any other commands and produce the message \texttt{SyntaxError} with the offending line and a ^ pointing to the part in the line with the error.
- Game: spot the syntax errors!

```
1 print("hello")
2 x = 0
3 while True
4    x = x + 1
5 mylist = ["bob" 2, False]
6 if x < 1:
7   print("x less than 1")
```
Exceptions: “Colorless green ideas sleep furiously”³

▶ If you follow all the syntax rules, the interpreter will try to execute your code.
▶ However, the interpreter may run into code it doesn’t know how to handle so it raises an Exception.
▶ The program has to deal with this Exception if it is not handled, execution aborts.
▶ Note: unlike with syntax errors, all the instructions before the interpreter reaches an exception do execute.

³Noam Chomsky (1955)
There are many types of exceptions, and eventually you will also be able to define your own exceptions.

I’ll show you some examples of common Exceptions.

```
x = 6
y = x / (x - 6) #syntax is OK, executing fails
```

File "test.py", line 2, in <module>
y = x / (x - 6)
ZeroDivisionError: integer division or modulo by zero
Exceptions: NameError

- Raised when the interpreter cannot find a name-binding you are requesting.
- Usually happens when you forget to bind a name, or you are trying to access a name outside your namespace.

```python
def foo():
    x = "hello"
foo()
print(x)
```

```
Traceback (most recent call last):
  File "exceptions.py", line 4, in <module>
    print(x)
NameError: name 'x' is not defined
```
Exceptions: IndexError

- Raised when the interpreter tries to access a list index that does not exist

```python
mylist = ["bob", "alice", "nick"]
print(mylist[len(mylist)])
```

```
Traceback (most recent call last):
  File "exceptions.py", line 2, in <module>
    print(mylist[len(mylist)])
IndexError: list index out of range
```
Exceptions: TypeError

- Raised when the interpreter tries to do an operation on a non-compatible type.

```python
>>> mylist = ["bob", "alice", "nick"]
>>> mylist + "mary"
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
```

```python
# this is okay
>>> mylist * 2
["bob", "alice", "nick", "bob", "alice", "nick"]
```
Some notes on exceptions

- Exceptions are just objects with names.
- The ones I have shown you are pre-defined for you (built-in)
- Syntax errors are also exceptions.
- Here is a list of all the built-in exceptions and some info on them.

```
>>> id(IndexError)
4296340480
>>> dir(IndexError)
['__class__', '__delattr__', ... '__subclasshook__', '__unicode__', 'args', 'message']
```
Traceback

- When an exception is raised, you get a traceback message which tells you where the error was raised.

```python
def foo():
    return 5 / 0
def fee():
    return foo()
fee()

Traceback (most recent call last):
File "exception.py", line 5, in <module>
    fee()
File "exception.py", line 4, in fee
    return foo()
File "exception.py", line 2, in foo
    return 5 / 0
ZeroDivisionError: division by zero
```
Where do exceptions come from?

- Exceptions come from `raise` statements.
- Syntax: `raise [exception object]`
- You can choose to raise any exception object. Obviously a descriptive exception is preferred.
- You can even define your own exceptions but we leave this for a later lecture.

```python
def my_divide(a, b):
    if b == 0:
        raise ZeroDivisionError
    else:
        return a / b
```
Handling Exceptions

- When an exception is raised, the exception is passed to the calling block.
- If the calling block does not handle the exception, the program terminates.

```python
#unhandled exception
def list_divide(numerators, denominators):
    ratio = []
    for a, b in zip(numerators, denominators):
        ratio.append(my_divide(a, b))
    return ratio
list_divide([1, 2, 1, 0], [1, 1, 0, 2])
```

**Life Hack 1**

The `zip(*args)` function lets you iterate over lists simultaneously. Yields tuple at each iteration with \((a[i], b[i])\).
try and catch

- Python executes the `try` block.
- If the code inside the `try` raises an exception, python executes the `except` block.

```python
# exception handled by caller
def list_divide(numerators, denominators):
    ratio = []
    for a, b in zip(numerators, denominators):
        try:
            ratio.append(my_divide(a, b))
        except ZeroDivisionError:
            print("division by zero, skipping")
            continue
    return ratio
list_divide([1, 2, 1, 0], [1, 1, 0, 2])
```
Try/catch: a more realistic example

- Often exceptions are caused by external users giving the program data it is not expecting.

```python
#not handling exceptions
while True:
    #if user gives invalid input program crashes
    x = int(input("Give me a number: "))

#handling exceptions
while True:
    try:
        x = int(input("Give me a number: "))
        break
    except TypeError:
        print("Not a number! Try again.")
```
Try/except/else: when no exception occurs

- An `else` block after a try/catch executes **only** if the `try` does not cause an exception.

```python
while True:
    try:
        a = int(input("Give me a numerator: "))
        b = int(input("Give me a denominator: "))
    except:
        print("Not a number! Try again.")
    else:
        print(f"{a} divided by {b} is {my_divide(a, b)}")
    break
```
Why not just do this?

```python
while True:
    try:
        a = int(input("Give me a numerator: "))
        b = int(input("Give me a denominator: "))
        print(f"{a} divided by {b} is {my_divide(a, b)}")
        break
    except:
        print("Not a number! Try again.")
```
And finally, the **finally** statement

- The **finally** block **always** executes after the **try** and before the **except**

- **Useful when:**
  1. The **except** or **else** block itself throws an exception.
  2. The **try** throws an unexpected exception.
  3. A control flow statement in the **except** skips the rest.

- **Why is it useful?** Often there are statements you **need** to perform before your program closes.
```python
while True:
    try:
        a = int(input("Give me a numerator: "))
        b = int(input("Give me a denominator: "))
    except:
        print("Not a number! Try again.")
        break
    else:
        result = my_divide(a, b)
    finally:
        print("hello from finally!")
        print("hello from the other siiiiiide")
```
Logical errors

- When according to Python your code is fine and runs without errors but it does not do what you intended.

- Example: spot the logical error

```python
#1
def my_max(mylist):
    for bla in mylist:
        my_max = 0
        if bla > my_max:
            my_max = bla
    return my_max
```

- There’s nothing to do to avoid logical errors other than testing your code thoroughly and having a good algorithm.

- Logical errors are often silent but deadly.