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

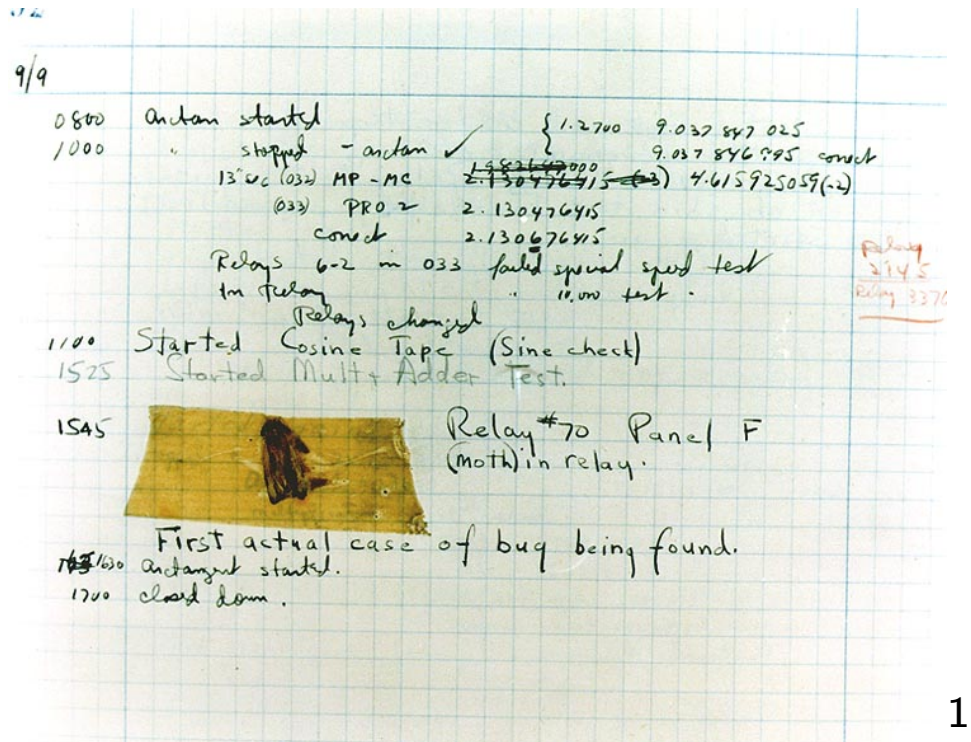
- Anatomy of a function, namespaces
- Importing

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.



[1]

[1] Wikipedia

Types of bugs

There are three major ways your code can go wrong.

1. Syntax errors
2. Exceptions (runtime)
3. Logical errors

Syntax Errors: "Furiously sleep ideas green colorless."[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 `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")
```

[2] Noam Chomsky (1955)

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.

[3] Noam Chomsky (1955)

Exceptions: ZeroDivisionError

- 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.
- `ZeroDivisionError`

```
1  x = 6
2  y = x  / (x - 6) #syntax is OK, executing fails
3
4  File "test.py", line 2, in <module>
5  y = x / (x - 6)
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.

```
1 def foo():
2     x = "hello"
3 foo()
4 print(x)
5 Traceback (most recent call last):
6   File "exceptions.py", line 4, in <module>
7     print(x)
8 NameError: name 'x' is not defined
```

Exceptions: IndexError

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

```
1 mylist = ["bob", "alice", "nick"]
2 print(mylist[len(mylist)])
3
4 Traceback (most recent call last):
5   File "exceptions.py", line 2, in <module>
6     print(mylist[len(mylist)])
7 IndexError: list index out of range
```

Exceptions: TypeError

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

```
>>> 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

# 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.

```
    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.

```
1  def my_divide(a, b):
2      if b == 0:
3          raise ZeroDivisionError
4      else:
5          return a / b
6  def my_divide(a, b):
7      if b == 0:
8          raise TypeError
9      else:
10         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.

```
1  #unhandled exception
2  def list_divide(numerators, denominators):
3      ratio = []
4      for a, b in zip(numerators, denominators):
5          ratio.append(my_divide(a, b))
6      return ratio
7  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.

```
1  #exception handled by caller
2  def list_divide(numerators, denominators):
3      ratio = []
4      for a, b in zip(numerators, denominators):
5          try:
6              ratio.append(my_divide(a, b))
7          except ZeroDivisionError:
8              print("division by zero, skipping")
9              continue
10     return ratio
11 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.

```
1  #not handling exceptions
2  while True:
3      #if user gives invalid input program crashes
4      x = int(input("Give me a number: "))
5  #handling exceptions
6  while True:
7      try:
8          x = int(input("Give me a number: "))
9          break
10     except TypeError:
11         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.

```
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?

```
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` trows 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.

`finally` example

```
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 siiiiide")
```

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

```
1  #1
2  def my_max(mylist):
3      for bla in mylist:
4          my_max = 0
5          if bla > my_max:
6              my_max = bla
7      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.