[220] Error Handling

Meena Syamkumar
Mike Doescher

Fill out exam 2 conflict form by Friday Oct 30th

Learning Objectives Today

How to crash more

- turn semantic bugs into runtime bugs with assert

How to crash less

- **assert** statements
- catch exceptions with **try/except**
- **raise** custom exceptions

https://en.wikipedia.org/wiki/Pizza

https://en.wikipedia.org/wiki/Grace_Hopper

Example: Pizza Analyzer

```
import math

def pizza_size(radius):
    return (radius ** 2) * math.pi

def slice_size(radius, slice_count):
    total_size = pizza_size(radius)
    return total_size * (1 / slice_count)

def main():
    for i in range(10):
        # grab input
        args = input("Enter pizza diameter(inches), slice count): ")
        args = args.split(',')
        radius = float(args[0].strip()) / 2
        slices = int(args[1].strip())

        # pizza analysis
        size = slice_size(radius, slices)
        print('PIZZA: radius={}, slices={}, slice square inches={}'
              .format(radius, slices, size))

main()
```

Exercise: what are possible bad inputs for

- diameter
- slice count
- other?

Does it cause a runtime error or semantic error?

Assert

Syntax:

```
assert BOOLEAN_EXPRESSION
```

Purpose:

Force program to crash if something is non-sensible, rather than run and produce garbage.

semantic errors
(hard to debug)

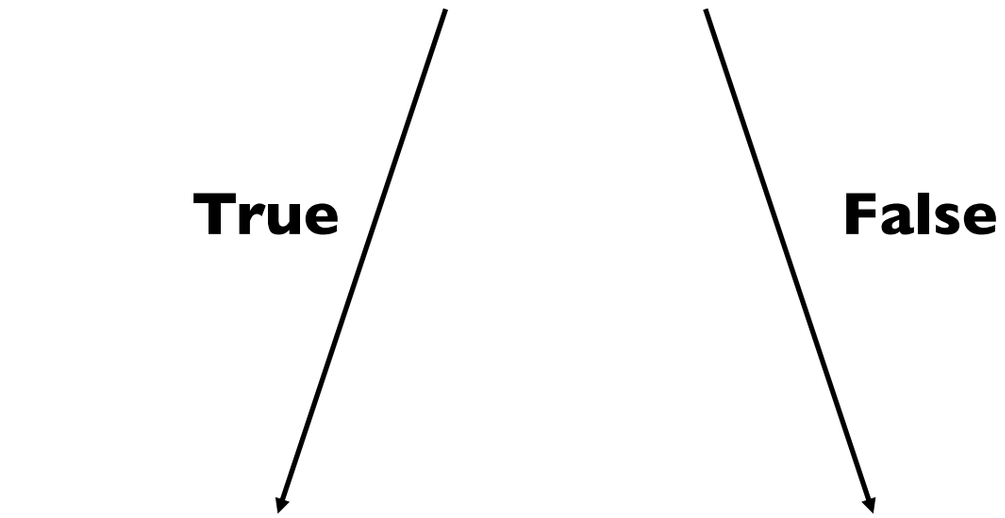
runtime errors
(easier to debug)

Assert

Warning: sometimes people disable assertions when running your code to improve performance

Syntax:

```
assert BOOLEAN_EXPRESSION
```



nothing happens

Crash!

```
Enter pizza diameter(inches), slice count): -10, 8
Traceback (most recent call last):
  File "pizza.py", line 24, in <module>
    main()
  File "pizza.py", line 20, in main
    size = slice_size(radius, slices)
  File "pizza.py", line 8, in slice_size
    total_size = pizza_size(radius)
  File "pizza.py", line 4, in pizza_size
    assert(radius > 0)
AssertionError
```

Assert

Syntax:

```
assert BOOLEAN_EXPRESSION
```

Examples:

```
assert x > 0

assert items != None

assert “age” in person

assert len(nums) % 2 == 1
```

Pizza Example: add asserts to crash upon

- diameter <= 0
- slices <= 0

What if we want to keep running even if there is an error?

Try/Except

Syntax:

```
flaky_function()
```

Try/Except

Syntax:

```
try:
    flaky_function()
except:
    print(“error!”) # or some other handling
```

Try/Except

Pizza Example: try/except to continue running upon

- parse errors
- analysis errors

Syntax:

```
try:
    flaky_function()
except:
    print(“error!”) # or some other handling
```

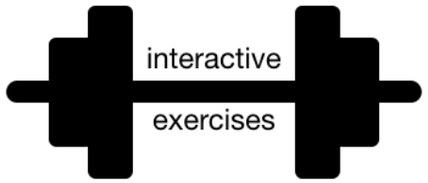
Description:

try and **except** blocks come in pairs (runtime errors are “exceptions”)

Python tries to run the code in the **try** block. If there’s an exception, it just runs the **except** block (instead of crashing). This is called **“catching”** the exception.

If there is no exception, the **except** block does not run.

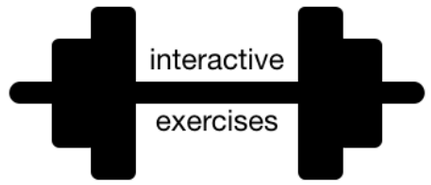
Exceptions are Exceptions to Regular Control Flow




```
try:
    print("2 inverse is", 1/2)
    print("1 inverse is", 1/1)
    print("0 inverse is", 1/0)
    print("-1 inverse is", -1/1)
    print("-2 inverse is", -1/1)
except:
    print("that's all, folks!")
```

never runs

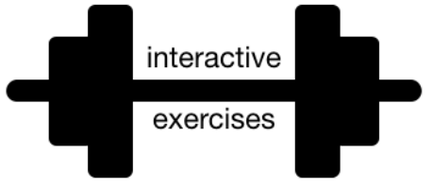
Exceptions are Exceptions to Regular Control Flow




```
def buggy():
    print("buggy: about to fail")
    print("buggy: infinity is ", 1/0)
    print("buggy: oops!") # never prints

def g():
    print("g: before buggy")
    buggy()
    print("g: after buggy") # never prints

def f():
    try:
        print("f: let's call g")
        g()
        print("f: g returned normally") # never prints
    except:
        print("f: that didn't go so well")

f()
```

Try this example in notebook or script format

Exceptions are Exceptions to Regular Control Flow




```
def buggy():
    print("buggy: about to fail")
    print("buggy: infinity is ", 1/0)
    print("buggy: oops!") # never prints

def g():
    print("g: before buggy")
    try:
        buggy()
    except:
        print("g: caught an exception from buggy")
    print("g: after buggy")

def f():
    try:
        print("f: let's call g")
        g()
        print("f: g returned normally")
    except:
        print("f: that didn't go so well")

f()
```

g catches, so f never knows about the exception

Try this example in notebook or script format

What if we want to know the reason for the exception?

Crash Cause

Pizza Example: print failure reasons
- for parse errors
- for analysis errors

Version 1:

```
try:
    flaky_function()
except:
    print(“error!”) # or some other handling
```

Version 2:

```
try:
    flaky_function()
except Exception as e:
    print(“error because:”, str(e))
```

e is of type Exception (very general)
(there are different types of exceptions)

get exception object
describing the problem

why it failed

What if we only want to catch certain exceptions?

Narrow Catching

Pizza Example: catch only real parse errors

- strings when want ints
- not enough values
- NOT typos in variable names

Version 2:

```
try:
    flaky_function()
except Exception as e:
    print("error because:", str(e))
```

Version 3:

only catch these two
(not NameError and others)

```
try:
    flaky_function()
Except (ValueError, IndexError) as e:
    print("error because:", str(e))
```

General Rule: always catch specific types of exceptions,
and/or make sure the user knows there was an error
(unexpected silent errors are the worst!)

Exception Hierarchy

Documentation:
https://docs.python.org/3/library/exceptions.html#exception-hierarchy

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

screenshot of hierarchy

What if we want to produce a specific kind of error? (not just an assert)

Custom Errors

```
BaseException
 +-- Exception
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
 +-- TypeError
 +-- ValueError
```

Asserts vs. Raising Exception Objects

Version 1 (quick and dirty):

```
def slice_size(radius, slice_count):
  assert radius > 0
  return pizza_size(radius) / slice_count
```

Version 2 (more robust and informative):

```
def slice_size(radius, slice_count):
    if radius <= 0:
       raise ArithmeticError("number of slices <= 0")
    return pizza_size(radius) / slice_count
```

tell Python this exception occurred here

create TypeError object

with this message

Summary

Asserts

- force a crash/exception
- better to crash in an obvious way than to use corrupt data

Exceptions

- produce them with raise
- catch them with try/except
- can choose specific types of exceptions

General Rule: always catch specific types of exceptions,
and/or make sure the user knows there was an error
(unexpected silent errors are the worst!)