Principles of Programming Languages

Lesson # 11

Exceptions

Exceptions

- Programmers must be always mindful of **possible errors** (different from BUGs!):
  - a function may not receive certain arguments,
  - a necessary resource may be missing,
  - a connection across a network may be lost

- Programmer must:
  1. *anticipate* the exceptional circumstances that may arise,
  2. take appropriate measures to *handle* them

There is no single correct approach

Shopping, weather forecast, search engine, ...

- ***Web server*** should be robust to errors, logging them for later consideration but *continuing to service new requests as long as possible*
- ***Python interpreter*** handles errors by terminating immediately and *printing an error message*

- Programmers must make *conscious choices* about **how** their programs should *react* to exceptional conditions

Exceptions

- Provide a **general mechanism** for adding **error-handling logic** to programs.

- **Separates** between normal code and the code that handles errors (runs only if something exceptional happen)

Exceptions

- ***Raising an exception*** is a technique for
  - interrupting the normal flow of execution in a program,
  - signaling that some exceptional circumstance has arisen,
  - returning directly to an enclosing part of the program that was designated to react to that circumstance.

Interpreter vs. user programs

- The Python interpreter *raises an exception* **each time** it detects an error in an expression or statement

Using built-in functions

- Users can also raise exceptions with **raise** and **assert** statements.

User-defined functions

Raising exceptions

- An ***exception*** is a object instance of a class that inherits, either directly or indirectly, from the **BaseException** class

- The ***assert*** statement raises an exception with the class **AssertionError**

- ***Any exception instance*** can be raised with the ***raise*** statement.

Common use

The most common use of raise:

1. constructs an exception instance and
2. raises it

```
>>> raise Exception(’An error occurred’)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
Exception: an error occurred
```

Flow

- When an exception is raised, no further statements in the current block of code are executed

- Unless the exception is handled, the interpreter will:
  1. Print a **stack backtrace** - a structured block of text that describes the nested set of active function calls in the branch of execution in which the exception was raised;
  2. Return directly to the interactive **read-eval-print** loop

Example

```
>>> raise Exception(’An error occurred’)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
Exception: an error occurred
```

<stdin> indicates that the exception was raised by the user in an interactive session, rather than from code in a file

Also can be raised by built-in functions

- Examples:
  - ZeroDivisionError: division by 0
  - IndexError: index out of range
  - I/O exceptions: file does not exist, etc.

Handling exceptions

An exception can be handled by an enclosing **try statement** that consists of multiple clauses:

- the first begins with **try** and
- the rest begin with **except**:

```
try:
        <try suite>
except <exception class> as <name>:
        <except suite>
...
```

Normal code, runs always

Runs only if something is wrong

Handling exceptions

- The **<try suite>** is always executed

- **<except suite>** is only executed *when an exception is raised* during the course of executing the <try suite>

- Each **except clause** specifies the particular class of exception to handle

Example

- If the **<exception class>** is **AssertionError**, then *any instance of a class inheriting from AssertionError* raised during the executing the <try suite> will be handled by the following **<except suite>**

- Within the <except suite>, the identifier **<name>** is bound to the exception object that was raised (only inside the <except suite>)

Example

We can handle a **ZeroDivisionError** exception using a try statement that binds the name x to 0 when the exception is raised

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print(’handling a’, type(e))
        x = 0
handling a <class ’ZeroDivisionError’>
>>> x
0
```

Control flow with exceptions

- A try statement will handle exceptions that occur within the body of a function that is applied within the **<try suite>**

- When an exception is raised, control jumps directly to the body of the **<except suite>** of the try statement that handles that type of exception

Example

```
>>> def invert(x):
        result = 1/x # Raises a ZeroDivisionError if x is 0
        print(’Never printed if x is 0’)
        return result
>>> def invert_safe(x):
        try:
            return invert(x)
        except ZeroDivisionError as e:
            return str(e)
>>> invert_safe(2)
Never printed if x is 0
0.5
>>> invert_safe(0)
’division by zero’
```

Exception Objects

- Exception objects themselves carry attributes, such as
  1. the **error message** stated in an assert statement
  2. information about **where** in the course of execution the exception was raised

- User-defined exception classes can carry additional attributes

Example: Newton’s method

```
>>> def approx_derivative(f, x, delta=1e-5):
        df = f(x + delta) - f(x)
        return df/delta
>>> def newton_update(f):
        def update(x):
            return x - f(x) / approx_derivative(f, x)
        return update
>>> def find_root(f, initial_guess=10):
        def test(x):
            return approx_eq(f(x), 0)
        return iter_improve(newton_update(f), test, initial_guess)
```

Use example

```
>>> def square_root(a):
        return find_root(lambda x: square(x) - a)
>>> square_root(16)
4.000000000026422
```

$2x^2 + \sqrt{x}.$ = a

```
>>> def func_root(a):
        return find_root(lambda x: 2*x*x + sqrt(x) - a)
```

$2x^2 + \sqrt{x}.$

```
>>> func_root(0)
???
```

fail to return any guess of the zero

```
>>> def iter_improve(update, test, guess=1):
        print(guess)
          while not test(guess):
            guess = update(guess)
            print(guess)
        return guess

>>> func_root(0)
```

```
10
4.940943260509369
2.3870712267378624
1.0761585432683334
0.37553813282078274
-0.010501189601387517
Traceback (most recent call last):
  File "C:\Users\marinal\Documents\TEACHING\PPL\Lectures\2014-2015\iter-improve-math-
error.py", line 30, in <module>
    func_root(0)
  File "C:\Users\marinal\Documents\TEACHING\PPL\Lectures\2014-2015\iter-improve-math-
error.py", line 18, in func_root
    return find_root(lambda x: 2*x*x + sqrt(x) - a)
  File "C:\Users\marinal\Documents\TEACHING\PPL\Lectures\2014-2015\iter-improve-math-
error.py", line 15, in find_root
    return iter_improve(newton_update(f), test, initial_guess)
  File "C:\Users\marinal\Documents\TEACHING\PPL\Lectures\2014-2015\iter-improve-math-
error.py", line 25, in iter_improve
    while not test(guess):
  File "C:\Users\marinal\Documents\TEACHING\PPL\Lectures\2014-2015\iter-improve-math-
error.py", line 14, in test
    return approx_eq(f(x), 0)
  File "C:\Users\marinal\Documents\TEACHING\PPL\Lectures\2014-2015\iter-improve-math-
error.py", line 18, in <lambda>
    return find_root(lambda x: 2*x*x + sqrt(x) - a)
ValueError: math domain error
```

Example: improved Newton's method

- A math domain error (a type of **ValueError**) is raised when *sqrt* is applied to a *negative number*

- Define an ***exception class*** that returns the best guess discovered in the course of iterative improvement whenever a **ValueError** occurs
  - **IterImproveError** that stores the *most recent guess* as an *attribute*
  - We'll handle this exception by raising its instance

First step

Define a new class that inherits from Exception:

```
>>> class IterImproveError(Exception):
        def __init__(self, last_guess):
            self.last_guess = last_guess
```

Second step

define a version of IterImprove that handles any **ValueError** by raising an **IterImproveError** that stores the most recent guess:

```
>>> def iter_improve(update, test, guess=1):
        try:
            while not test(guess):
                guess = update(guess)
            return guess
        except ValueError:
            raise IterImproveError(guess)
```

BEWARE: Negative number!!!

Final step

find_root handles an IterImproveError by returning its last guess:

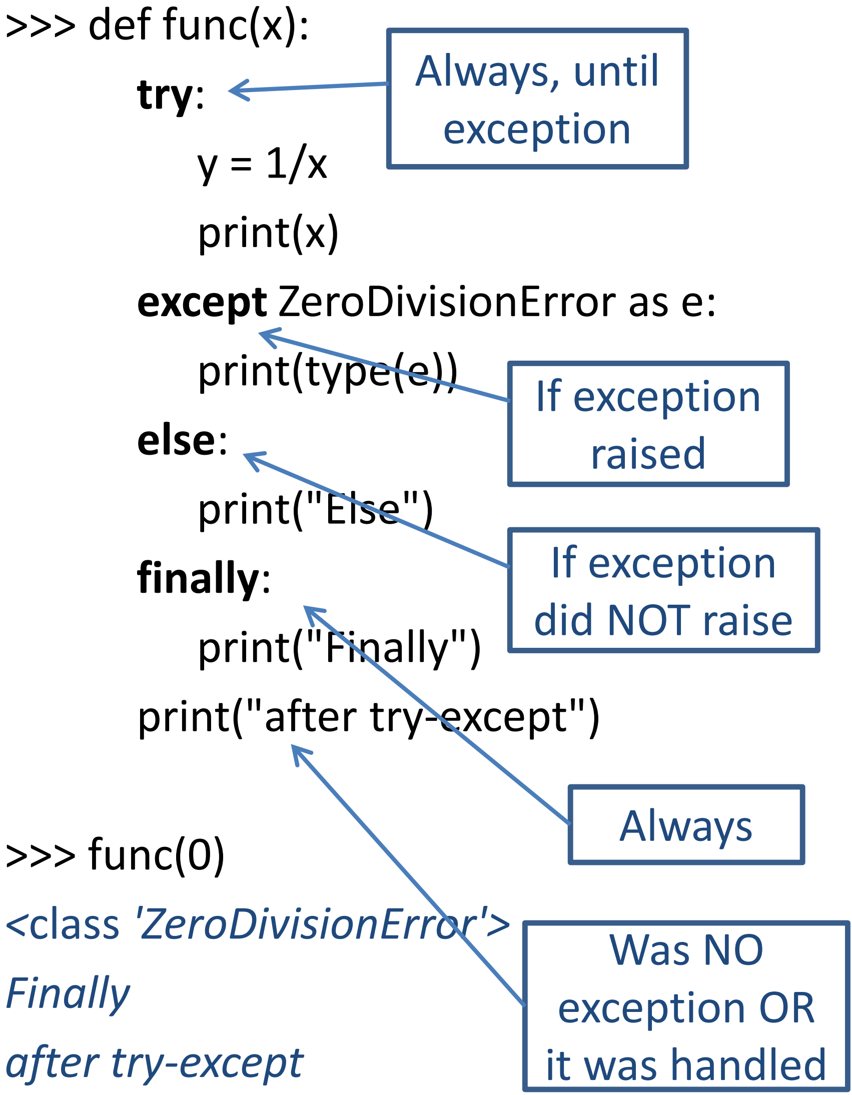
```
>>> def find_root(f, initial_guess=10):
        def test(x):
            return approx_eq(f(x), 0)
        try:
            return iter_improve(newton_update(f), test, initial_guess)
        except IterImproveError as e:
            return e.last_guess
```

Example

- Apply **find_root** to find the zero of the function $2x^2 + \sqrt{x}.$
- Evaluating it on any negative number will raise a **ValueError**
- Returns the last guess found before the error

```
>>> from math import sqrt
>>> func_root(0)
-0.010501189601387517
```

More examples



```
>>> def func(x):
        try:
            y = 1/x
            print(x)
        except ZeroDivisionError as e:
            print(type(e))
        else:
            print("Else")
        finally:
            print("Finally")
        print("after try-except")
```

Always, until exception

If exception raised

If exception did NOT raise

Always

Was NO exception OR it was handled

```
>>> func(0)
<class 'ZeroDivisionError'>
Finally
after try-except
```

```
>>> func(3)
3
Else
Finally
after try-except
```

```
>>> func("a")
Finally
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    func("a")
  File "<pyshell#18>", line 3, in func
    y = 1/x
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

What do we need it for?

- Exceptions are another technique that help us to separate the concerns of our program into *modular parts*

- Example: Python’s exception mechanism allowed us to separate the *logic for iterative improvement*, from the *logic for handling errors*

To be continued…

- We will also find that exceptions are a very useful feature when implementing interpreters in Python