Dealing with Exceptional Situations

So far, we have assumed input to functions is valid. In this lecture, we show you what to do when input is invalid (or preconditions are not met).

Errors

We have all seen errors!

Division by 0

Calling `1 / 0` results in the following:

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in
    1 / 0
ZeroDivisionError: division by zero
```

Value Error

Calling `'abc'.index('q')` or `int('moogah')` results in the following:

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in
    'abc'.index('q')
ValueError: substring not found
```

and

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in
    'int('moogah')
ValueError: invalid literal for int() with base 10: 'moogah'
```

Both are value errors, but with very different details.

You can call for help on errors (i.e. `help(ValueError)`)

Errors happen when the code tries to make a function do something it wasn't designed to do.
You can use exceptions when you encounter a situation that is not part of the normal flow of execution.

Exception Handling

Python has an exception handling statements:

The `try` statement (simple form):

```
try:
    statements
except:
    statements
```

If a statement in the `try` block raises an exception, the remaining statements in the block are not executed.

With the `except` clause, we can specify the exception we would like to catch.

For example,

```
try:
    1 / 0
except ZeroDivisionError:
    print("Divided by zero.")
```

This code still catches the divide by zero error; however,

```
try:
    1 / 0
except ValueError:
    print("Divided by zero.")
```

will not catch the error.

Python can handle both errors separately.

`try` statement (specifying exception types):

```
try:
    statements
except ExceptionType:
    statements
except ExceptionType:
    statements
```

With multiple `except` clauses, Python executes the block of the first matching exception type.

Warning: If you want to handle two kinds of exceptions, and one is a subclass of the other, catch the subclass first.

Every exception is an object and can be assigned a variable name. This variable can be used like any other variable.

Raise Errors

We can cause an exception to happen by using the `raise` statement.

For example:

```
raise ValueError("Oops. Bad value. No cookie.")
```

results in:

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in
    raise ValueError("Oops. Bad value. No cookie.")
ValueError: Oops. Bad value. No cookie.
```

Example Program

```
def raise_an_exception(v):
    raise ValueError("{} is not a valid value.".format(v))

def main():
    raise_an_exception(3)

if __name__ == '__main__':
    try:
        main()
    except ValueError as ve:
        print(ve)
```

Assertion

Programmers often use the `assert` statement to ensure that preconditions are met.

For example:

```
def every_nth(L, n=1):
    """ (list, int) -> list

    Precondition: 0 <= n < len(L)

    Return a list containing every nth item of L,
    starting at index 0.

    >>> every_nth([1, 2, 3, 4, 5, 6], n=2)
    [1, 3, 5]
    >>> every_nth([1, 2, 3, 4, 5, 6], 3)
    [1, 4]
    >>> every_nth([1, 2, 3, 4, 5, 6])
    [1, 2, 3, 4, 5, 6]
    """

    assert 0 <= n < len(L), '{} is out of range.'.format(n)

    result = []

    for i in range(0, len(L), n):
        result.append(L[i])

    return result

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

This program runs without errors if the preconditions are met, but raises an error if the conditions are not met.

Tips

Tips for when and how to use exceptions:

- Exceptions should **not** be used as part of the normal flow of a program. They are for exceptional situations.
- Only catch exceptions that you know how to handle.
- "Throw low, catch high."

- Use an `assert` statement to verify that preconditions are met.

Jennifer Campbell • Paul Gries
University of Toronto
