CIS192 Python Programming

Iterators, Generators, IO, and Exceptions

Harry Smith

University of Pennsylvania

February 15, 2018

Outline

What's a For Loop?

- What's going on here:

```
for num in [1, 2, 3]:
    print num
```

Iterables

- An iterable can be for-looped over, and is:
  - an object that supports `__iter__` returns iterator or `__getitem__` for indexed lookup
  - e.g. string, list, tuple, file
- Wait, what's an iterator?
  - returns the next item from calls to `__next__()`
  - raises `StopIteration` if `__next__()` called too many times

Iterators

Iterators in action:

```
>>> i = iter(l)
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
>>> next(i)
StopIteration
```

Iterators

`for ... in ...` is just syntactic sugar for the following:

1. Call `iter(...)` to create an iterator
2. Call `next` on the iterator
3. Catch `StopIteration` exceptions

Generators

- A generator is a function that behaves like an iterable, but without all the extra boilerplate and extra state
- `next(generator)` will execute the function body until yield is reached
- `yield` is like `return`, except that the state is remembered
- Reaching the end of the function raises `StopIteration`

Generator Comprehensions

- `(x ** 2 for x in [1, 2, 3])#Not a tuple comprehension!`
- A generator comprehension creates a generator object
- `g = ({expr} for x in iterable)` is equivalent to:

```
def g():
  for x in iterable:
    yield {expr}
```

Why Generators?

- Memory Efficient
  - Keep 1 value in memory at a time
  - The function state is minimal in terms of memory
  - Use a generator over a list whenever you iterate
  - Great: `for square in (x ** 2 for x in range(10000))`
  - Not so great:
    ```
    for square in [x ** 2 for x in range(10000)]
    ```
- Good for aggregating (summing, counting) items
- Good for infinite sequences
- Bad if you need to inspect the individual values

Infinite Generators

- Infinite computations...
  - `itertools.count()` is an infinite generator
  - `itertools.islice()` lets you slice iterables

Raise Exceptions

- An exception can be raised with the `raise` keyword
- Raising an exception sends control back up to the nearest enclosing exception handler
- What if the exception isn't handled?
  - The interpreter prints a stack trace
  - The program exits or returns to the interactive loop

Exceptions

- `KeyError`: accessing a non-existent dictionary key
- `AttributeError`: calling a non-existent method
- `NameError`: referencing a non-existent variable
- `TypeError`: mixing data-types
- `ValueError`: right type, wrong value
- `ImportError`: module not available
- `IOError`: file does not exist

Exceptions

- Syntax:
  - Java: `try...catch` to handle, and `throw` to generate
  - Python: `try...except` to handle, and `raise` to generate
- When do I use `try...except`?
  - Opening a file (may not exist)
  - User input (never trust anybody)
  - Connecting to a database (might be unavailable)

Catching Exceptions

Catch all exceptions, regardless of type:

```
def int_default_0(x):
    try:
        return int(x)
    except:
        return 0

>>> int_default_0('5')
5
>>> int_default_0('hi')
0 # would have thrown a ValueError
>>> int_default_0([])
0 # would have thrown a TypeError
```

Catching Exceptions

Catch only a specific type:

```
def int_default_0(x):
    try:
        return int(x)
    except ValueError:
        return 0

>>> int_default_0('hi')
0
>>> int_default_0([])
TypeError
```

Catching Exceptions

Catch multiple types together:

```
def int_default_0(x):
    try:
        return int(x)
    except (ValueError, TypeError):
        return 0

>>> int_default_0('hi')
0
>>> int_default_0([])
0
```

Catching Exceptions

Multiple except blocks:

```
def int_default_0(x):
    try:
        return int(x)
    except ValueError:
        print "Caught a ValueError."
    except TypeError:
        print "Caught a TypeError."

>>> int_default_0('hi')
Caught a ValueError.
>>> int_default_0([])
Caught a TypeError.
```

Catching Exceptions

Get a reference to the Exception class instance:

```
def int_default_0(x):
    try:
        return int(x)
    except (ValueError, TypeError) as e:
        print(e)
        return 0

>>> int_default_0('hi')
invalid literal for int() with base 10: 'hi'
>>> int_default_0([])
int() argument must be a string or a number,
not 'list'
```

Catching Exceptions

- Enclose code that might throw an exception in a `try` block
- Specify an `except` block to be executed if an exception is raised
- It's best to specify specific errors with `except ExceptionType as name:`
- Catch any type of error with `except:`
- Include an `else` block if you need to do something when there isn't an error
- The `finally` block gets executed no matter what
- You can have multiple `except` clauses
- There must be at least 1 `except` clause or a `finally` clause

User Defined Exceptions

- Often inheriting from `Exception` is enough

```
class MyException(Exception)
    pass
```

- You can define other attributes
- Access those attributes when the exception is caught
- Implementing `__str__` and `__repr__` is also useful

Standard Input

- You can ask the user for input on STD_IN
- `input()` will read and return STD_IN up to a newline
- `input(prompt)` prints `str(prompt)` before reading input
- Standard In is accessible as a file-object: `sys.stdin`
- `print(string)` sends `string` to STD_OUT
- `print(s, end='')` prints without a trailing newline
- Standard In is accessible as a file-object: `sys.stdout`

File IO

- `open(name, mode)` returns a file-object
- `name` is the path of the file to open
- If `mode == 'r'`, the file is open in read-only mode
- If `mode == 'w'`, the file is open in write-only mode
  - `'w'` Truncates the file first
- If `mode == 'a'`, like `'w'` but appends to the file
- Supplying `'+'` after one of `'rwa'` is for reading and writing
  - Starting position in file depends on `'rwa'`
  - `'w'` still truncates

File Operations

- Given a file object `f = open(name, 'a+')`
- `f.readline()` reads a line
- `f.read()` reads the whole file (up to EOF)
- `f.write(string)` writes string without adding a newline
- `f.writelines(lines)` writes lines without adding newlines
- `f.flush()` flushes the write buffers
- `f.close()` flushes and closes the file
- `f.seek(offset)` sets the position in the file

With Statement

- `with expr as name:` begins a managed block
- Before the block is executed:
  - The `__enter__()` method of `expr` is called
  - The result is assigned to `name`
- The block is executed in a `try` block
- Any exceptions are passed to the `__exit__()` method of `expr`
- `__exit__(exc_type, exc_val, exc_trace_back)`
  - The arguments to `__exit__` can be used to handle certain errors
- `finally __exit__(None, None, None)` will be called

File With Statements

- It's good practice to always close files
- Remembering is hard...
- `with open(...)as f_name:`
- The `__enter__` and `__exit__` methods of file-objects make sure that the file gets closed

Take-aways

- Use a Generator if you don't need to have it all at once
- If something can fail $\rightarrow$ use a `try` block
- with statements can manage resources for you

Working Example

- How can we tie all of these concepts together?

Working Example: Motivation

Figure: A college radio station

Figure: A band needing an interview

Working Example: What do we need

- An interface for recording notes
- A way to keep track of timestamps
- A way to submit questions
- A file to store the answers