CIS192 Python Programming
Introduction

Robert Rand
University of Pennsylvania
September 16, 2015
1 Iterators, Generators, Exceptions, and IO
   - Iterators
   - Generators
   - Exceptions
   - Input Output
   - Context Managers
Iterators

- An **iterable** is an object which supports `__iter__()`
- `__iter__()` should return an object that:
  - returns the next item from calls to `__next__()`
  - raises `StopIteration` if `__next__()` called too many times
  - returns `self` from `__iter__()`
for x in iterable expands to calls to iter and next

An iterator is constructed: iter(iterable)

next() is called on that iterator

Values are bound to x

StopIteration is caught and the loop terminates
Generators

- A generator is a function that behaves like an iterable
- `next()` 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`
- A generator comprehension creates a generator object
- `g = (expr for x in iterable)` Translates:
  ```python
def g():
    for x in iterable:
      yield expr
  ```
Why use 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
  - Bad: `for x in [expr for y in iterable]`
  - Good: `for x in (expr for y in iterable)`

- **Incremental callbacks**
  - Yield updates as the function executes
Generators don’t need to ever return `StopIteration`

- `itertools.count` generates an infinite sequence of naturals
- `itertools.islice` takes a slice of the given generator

Built in higher-order generator functions:
- `itertools.imap` maps a function onto two potentially infinite generators
- `itertools.ifilter` applies a filter to a potentially infinite generator
Raise Exceptions

- An exception can be raised with the `raise` keyword.
- Raising an exception sends control back up to the nearest enclosing exception handler.
- If the exception is not handled:
  - The interpreter prints a stack trace.
  - The program exits or returns to the interactive loop.
Types of Exceptions

- **BaseException**: Don’t inherit directly from this
- **Exception**: Use this as the base class
- **AttributeError**: `obj.attribute` fails
- **IndexError**: invalid index to `seq[i]`
- **KeyError**: Failed dictionary look-up
- **StopIteration**: Raised in `next()` for iterators
- **TypeError**: Wrong type or number of arguments
- **ValueError**: Right type but wrong value
- **OSError**: system call errors (file not found)
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
  ```python
  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
  ```python
class MyException(Exception):
    pass
  ```
- You can define other attributes
- Access those attributes when the exception is caught
- Implementing `__str__` and `__repr__` is also useful
You can ask the user for input on STD_IN

*input()* will evaluate from STD_IN. Do Not Use!

*raw_input()* will read and return STD_IN up to a newline

*raw_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*
- `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'` and `'w'` still truncates
File Operations

- Given a file object `f = open(name, 'a+t')`
- `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 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 → use a try block
- with statements can manage resources for you