Python Programs

* Are made up of *modules*

* One module is the “main” (top-level) module
  * The *first one loaded* (even if it’s the interpreter)
  * Its module object has “__main__” as its __name__ attribute
  * It may load other modules (via *import*)

* Modules are made up of *statements*
Python Statements

- Assignments
- Control flow
  - if, while, for, break, continue, return, yield
- Exceptions
  - assert, try, raise
- Definitions
  - def (functions), class
- Namespace directives
  - import, global
- Miscellaneous
  - del, print, exec
Statements Can Be Placed Anywhere

```python
if debug:
    def square(x):
        if not isinstance(x, float):
            raise TypeError("Expected a float")
        return x * x
else:
    def square(x):
        return x * x

square is defined at the top (aka module) level
```
Loops and Iteration

- Loops are **for** and **while**

- Both can have an **else** clause
  - Executes only if the loop runs to *completion* (even an *empty* loop)
  - Does not execute if the loop if terminated with a **break** or call to **sys.exit()**

```python
def line in open("foo.txt"):
    stripped = line.strip()
    if not stripped:
        break
    # process the stripped line
    ...
else:
    raise RuntimeError("Missing section separator")
```
Getting Sequence Index Numbers

- Use `enumerate(<seq>)`

- Returns a `<index>,<item>` pair

```python
>>> s = ['a','b','c']
>>> iter = enumerate(s)
>>> iter
<enumerate object at 0x1004aab90>
>>> for i,x in iter: print i,x
... 0 a
1 b
2 c
>>> dict(enumerate(s,10))  # 2nd argument is starting index
{10: 'a', 11: 'b', 12: 'c'}
```
Exception Handling

```python
# except.py
import sys

try:
    #    exit(0)
    #    raise IOError('an IO message')
    #    raise Exception('a generic message')
    #    raise LookupError('Quick! Look up!')
    print 'No exception raised'
except IOError as e:
    print e
except Exception as e:
    print e
except:
    print sys.exc_info()
else:
    print 'No exception caught'
finally:
    print 'This always prints'
```

See Table 5.1 for the Exception Type Hierarchy

div class="rte">
else is handy for file processing
</div>
Exception Notes

* If an exception is raised but *not handled*, and a **finally** clause is present, the exception is automatically **re-raised** after the **finally** clause executes.

* You can **re-raise** the current exception from inside an **except** clause by executing **raise** *without an argument*.

* You can also raise a **new exception**:
  * The old one is lost.

* You can define your own exception types by **subclassing** another:
  * The constructor receives the argument(s) from the **raise** statement.

* You can get current exception info from **sys.exc_info()**.
Assertions

* Used for *debugging* (see next slide...)

* You place assertions in code for *invariants*
  * Conditions that should always be *True*
  * Enforcing *design decisions*, for example (preconditions, postconditions)

* If the condition is *False*, an *AssertionError* exception is raised
  * You have a *bug* when that happens
  * Do **NOT** use for checking user errors or doing actual work!

* Assertions can be *ignored* by running Python with the -O flag
Assertion Example

• Common use: Function **Preconditions**:

```python
class Stack(object):
    def __init__(self):
        self.data = []
    def push(self, x):
        self.data.append(x)
    def pop(self):
        assert len(self.data), "Stack underflow"
        self.data.pop()
    def top(self):
        assert len(self.data), "Stack underflow"
        return self.data[-1]
    def size(self):
        return len(self.data)

>>> stk = Stack()
>>> stk.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in pop
AssertionError: Stack underflow
```
**Assertion Example**

- **Common use:** *Class invariants*
  - Can be checked after construction and before and after each method call

```python
class Rational(object):
    def __init__(self, num = 0, den = 1):
        "Stores fraction in lowest terms, with denominator>0"
        assert den != 0
        common = Rational.gcd(num,den)
        if den < 0:
            num,den = -num,-den
        self.num, self.den = num/common, den/common
        assert Rational.gcd(num,den) == 1, "invariant violation"
```
Deterministic Destruction

* Refers to automatic cleanup that takes place when a variable goes out of scope
  * Releasing resources, such as memory, connections, etc.

* C++ does this with destructors

* Python uses try / except / else / finally, or context managers
  * Use the with ... as syntax
  * Defines a scope (context) for an object’s lifetime
Context Manager Example

- The **file** type defines a context
- When the **with** block is exited, the file is *automatically closed*, if needed

```python
>>> with open('except.py') as f:
...     for line in f:
...         print line,
... # Continue program (file is closed here)...
```

```python
>>> lock = threading.Lock()
>>> with lock:
>>>     # Critical section...
```
class File {
    FILE* f;
public:
    File(const char* fname) {
        f = fopen(fname,"r");  // open file
        if (!f)
            throw std::runtime_error("no such file");
        printf("%s opened\n", fname);
    }
    ~File() {
        fclose(f);              // close file
        puts("file closed");
    }
    int linecount() const {
        int ch, count = 0;
        while ((ch = fgetc(f)) != EOF)
            if (ch == '\n')
                ++count;
        return count;
    }
};
Evidence of Determinism

```c
int main() {
    File f("Strings.txt");
    printf("# lines = %d\n", f.linecount());
}

/* Output:
Strings.txt opened
# lines = 192
file closed
*/
```
Defining Context Managers

* Define the following functions:

  * __enter__(self,...)
    * Called by with

  * __exit__(self, exc_type, exc_value, exc_tb)
    * Last 3 args have exception info
    * Will be None if no exception occurred
A Python Version

class MyFile(object):
    def __init__(self,fname):
        self.fname = fname
        self.f = open(self.fname)
        print self.fname,'opened'
    def __enter__(self):
        print '__enter__'
        return self  # Target of “as”
    def __exit__(self,exc_type,exc_value,exc_tb):
        self.f.close()
        print self.fname,'closed'
        return True  # Exception handled (doesn't propagate)
    def linecount(self):
        count = 0
        for line in self.f:
            count += 1
        return count

with MyFile('Strings.txt') as f:
    print f.linecount()

'''Output:
Strings.txt opened
__enter__
193
Strings.txt closed
'''