The Python Language (Part 2)

Robert M. Dondero, Ph.D.
Princeton University

Objectives

- We will cover:
  - A subset of Python...
  - That is appropriate for COS 333...
  - Through example programs

2

Objectives

Continuing from the previous slide deck…

[see slide]

Agenda

- **Data types and operators**
- **Terminal I/O**
- Catching exceptions
- Statements
- Throwing exceptions

Data Types and Operators & Terminal I/O

The same example programs illustrate data types and operators and terminal I/O

Data Types and Operators

- See **circle1.py**
  - The job:
    - Read a circle's radius from stdin
    - Write the circle's diameter and circumference to stdout

```
$ python circle1.py
Enter the circle's radius:
5
A circle with radius 5 has diameter 10
and circumference 31.415927.
$ python circle1.py
Enter the circle's radius:
1
A circle with radius 1 has diameter 2
and circumference 6.283185.
$
```


Data Types and Operators

[see slide]

Code notes:

No variable declarations
Terminal input via input()
String literals: '…' or "…"
Data types: int, float
Type conversions via int() and float()
Assignment statement
Operators: * (multiplication), % (string formatting)
print() is variadic

Data Types and Operators

- Generalizing
  - Conversion functions: `int(obj)`, `float(obj)`, `bool(obj)`, `str(obj)`
  - String formatting: conversion specifications are as in C
  - Data types, operators, and terminal I/O...

Data Types and Operators

Data Type	Example Literals
int	1, 23, 3493, 01, 027, 06645, 0x1, 0x17, 0xDA5 **(no theoretical size limit)**
float	0., 0.0, .0, 1.0, 1e0, 1.0e0 **(corresponds to C/Java double)**
bool	False, True
str	'hi',"hi"

6

Data Types and Operators

[see slide]

Notes:

If an int is small enough, then Python implements it as four bytes
If an int is too large to fit into 4 bytes, then Python implements it as Java implements a BigInt object
Also recall the BigInt ADT from COS 217

When evaluating an expression that generates a new int object, Python automatically chooses the proper implementation

No int overflow!!!

Data Types and Operators

Operators	(Priority) Meaning
`{key:expr,…}, {expr,…}`	(1) **Dictionary** creation; **set** creation
`[expr,…]`	(2) **List** creation
`(expr,…)`	(3) **Tuple** creation, or just parentheses
`f(expr,…)`	(4) Function call
`x[startindex:stopindex]`	(5) Slicing (for sequences)
`x[index]`	(6) Indexing (for containers)
`x.attr`	(7) Attribute reference
`x**y`	(8) **Exponentiation**
`~x`	(9) Bitwise NOT
`+x, -x`	(10) Unary plus, unary minus
`x*y, x/y, x//y, x%y`	(11) Mult, div, truncating div, remainder (or string formatting)
`x+y, x-y`	(12) Addition, subtraction

7

Data Types and Operators

For reference

[see slide]

Notes:

{} creates a dict object, or rarely a set object
[] creates a list object
() creates a tuple object
x[startindex:stopindex] generates a slice of sequence x
    x could be a tuple, list, str
** is exponentiation
There are two division operators...

Data Types and Operators

Beware of division operators:

Expression	Value
5 // 2	**2**
5 / 2	2.5
float(5) / float(2)	2.5
4 / 2	2.0
float(4) / float(2)	2.0

Suggestion: use only boldfaced forms

And that's quite different from Python 2!!!

Data Types and Operators

Operator	(Priority) Meaning
`x<<i, x>>i`	(13) Left-shift, right-shift
`x&y`	(14) Bitwise AND
`x^y`	(15) Bitwise XOR
`x|y`	(16) Bitwise OR
`x<y, x<=y, x>y, x>=y`	(17) Relational
`x==y, x!=y`	(17) Relational
`x is y, x is not y`	(18) **Identity** tests
`x in y, x not in y`	(19) **Membership** tests
`not x`	(20) Logical NOT
`x and y`	(21) Logical AND
`x or y`	(22) Logical OR

9

Data Types and Operators

For reference

[see slide]

Notes:

is operator checks objects for identity

== operator checks objects for equality -- more on that later

in operator checks for membership in a collection (dict, list, or tuple)

The logical operators are more reasonable than they are in C and Java

Terminal I/O

Reading from stdin:

```
str = input()
str = input(prompt_str)

from sys import stdin
…
str = stdin.readline()
```

Terminal I/O

Writing to stdout:

```
print(str)
print(str, end='')
print(str1, str2, str3)
print(str1, str2, str3, end='')
```

Writing to stderr:

```
from sys import stderr
…
print(str, file=stderr)
print(str, end='', file=stderr)
print(str1, str2, str3, end='', file=stderr)
```

Agenda

- Data types and operators
- Terminal I/O
- **Catching exceptions**
- Statements
- Throwing exceptions

Catching Exceptions

The short story: exception handling in Python is essentially the same as it is in Java

However, it has come to my attention that many students entering COS 333 aren't comfortable with exception handling in Java

So let's cover exception handling in Python reasonably thoroughly…

Catching Exceptions

- Recall **circle1.py**
  - Problem: Doesn't handle bad data

```
$ python circle1.py
Enter the circle's radius:
xyz
Traceback (most recent call last):
  File "circle1.py", line 26, in <module>
    main()
  File "circle1.py", line 15, in main
    radius = int(line)
ValueError: invalid literal for int() with base 10: 'xyz'
$ python circle1.py
Enter the circle's radius:
Traceback (most recent call last):
  File "circle1.py", line 26, in <module>
    main()
  File "circle1.py", line 14, in main
    line = input("Enter the circle's radius:\n")
EOFError
$
```

13

Catching Exceptions

[see slide]

Code notes:

Suppose stdin contains a non-integer
int(line) throws an exception
Uncaught exception causes Python to:
Write a stack trace to stderr
Write the exception (converted to a str) to stderr
Exit

Suppose stdin contains no data
input(…) throws an exception
Uncaught exception causes Python to:
Write stack trace to stderr
Write exception to stderr
Exit

Catching Exceptions

- See **circle2.py**
  - The job:
    - Same as circle1.py, but…
    - Handles bad data

```
$ python circle2.py
Enter the circle's radius:
5
A circle with radius 5 has diameter 10
and circumference 31.415927.
$ python circle2.py
Enter the circle's radius:
xyz
Error: Bad or missing data
$ python circle2.py
Enter the circle's radius:
Error: Bad or missing data
$
```


Catching Exceptions

[see slide]

Code notes:

try statement with except clause

When int() fails, it throws an exception which is an object of class ValueError

ValueError is a subclass of Exception

When input() fails, it throws an exception which is an object of class EOFError

EOFError is a subclass of Exception

Catching Exceptions

- See **circle3.py**
  - The job:
    - Same as circle2.py, but…
    - Writes more specific error messages

```
$ python circle3.py
Enter the circle's radius:
5
A circle with radius 5 has diameter 10
and circumference 31.415927.
$ python circle3.py
Enter the circle's radius:
xyz
Error: Not an integer
$ python circle3.py
Enter the circle's radius:
Error: Missing integer
$
```

15

Catching Exceptions

[see slide]

Code notes:

try statement with multiple except clauses
Propagation of exception through except clauses
Exact matches

Catching Exceptions

- See **circle4.py**
  - The job:
    - Same as circle3.py but…
    - Propagates exceptions

```
$ python circle4.py
Enter the circle's radius:
5
A circle with radius 5 has diameter 10
and circumference 31.415927.
$ python circle4.py
Enter the circle's radius:
xyz
Error: Not an integer
$ python circle4.py
Enter the circle's radius:
Error: Missing integer
$
```

16

Catching Exceptions

[see slide]

Code notes:

Exceptions need not be caught immediately
Uncaught exception propagates upward through function call stack

Generalizing…

Catching Exceptions

```
try:
    stmt1
    stmt2
    stmt3
except ExceptionClass1:
    stmt4
    stmt5
    stmt6
except ExceptionClass2:
    stmt7
    stmt8
    stmt9
stmt10
stmt11
stmt12
```

Case 1:
Python executes *stmt1*, *stmt2, stmt3* successfully.

17

Catching Exceptions

[see slide]

If no exceptions are thrown, then...
Python executes statements 1, 2, 3, 10, 11, and 12

Catching Exceptions

```
try:
    stmt1
    stmt2
    stmt3
except ExceptionClass1:
    stmt4
    stmt5
    stmt6
except ExceptionClass2:
    stmt7
    stmt8
    stmt9
stmt10
stmt11
stmt12
```

Case 2:
stmt2 throws an exception that matches ExceptionClass1.

The exception **matches** *ExceptionClass1* if the class of the exception is *ExceptionClass1* or any subclass of *ExceptionClass1*

Catching Exceptions

[see slide]

If stmt2 throws an exception that matches ExceptionClass1, then…
Python executes statements 1, 2, 4, 5, 6, 10, 11, and 12

Catching Exceptions

```
try:
    stmt1
    stmt2
    stmt3
except ExceptionClass1:
    stmt4
    stmt5
    stmt6
except ExceptionClass2:
    stmt7
    stmt8
    stmt9
stmt10
stmt11
stmt12
```

Case 3:
stmt2 throws an exception that does not match ExceptionClass1, but does match ExceptionClass2.

19

Catching Exceptions

[see slide]

If stmt2 throws an exception that does not match ExceptionClass1 but does match ExceptionClass2, then…
Python executes statements 1, 2, 7, 8, 9, 10, 11, and 12

Catching Exceptions

```
try:
    stmt1
    stmt2
    stmt3
except ExceptionClass1:
    stmt4
    stmt5
    stmt6
except ExceptionClass2:
    stmt7
    stmt8
    stmt9
stmt10
stmt11
stmt12
```

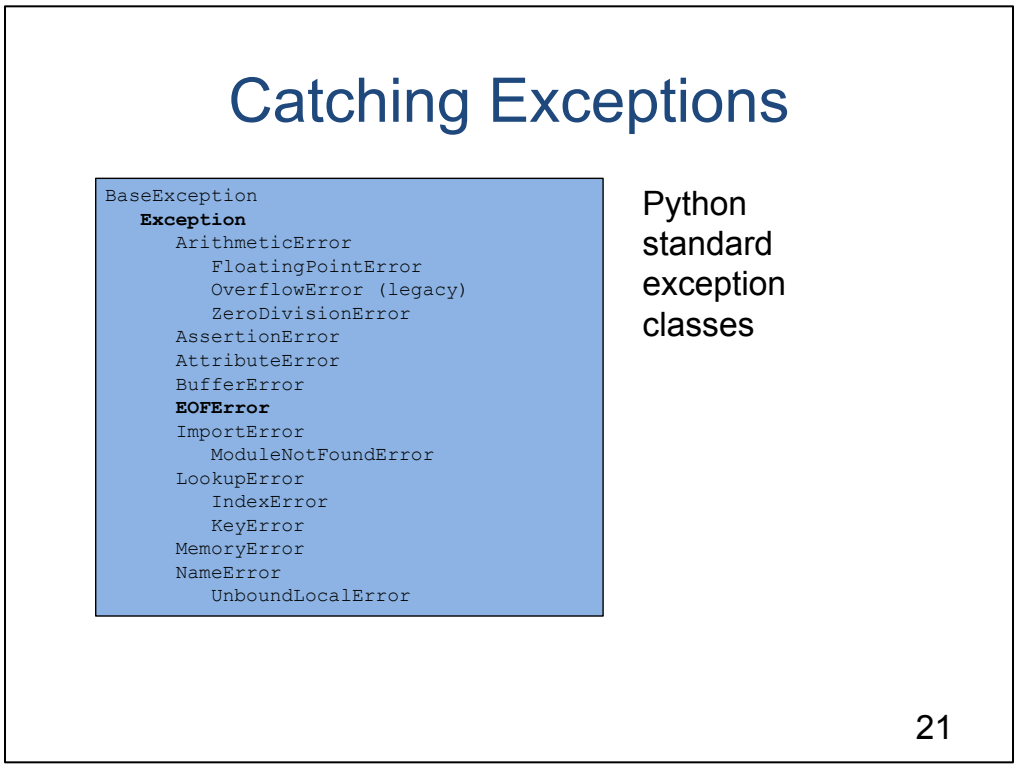
Case 4:
stmt2 throws an exception.
The exception does not match ExceptionClass1. The exception does not match ExceptionClass2.
Python propagates the exception up the call stack, and repeats the algorithm at each level

20

Catching Exceptions

[see slide]

If stmt2 throws an exception,
and the exception does not match ExceptionClass1
and the exception does not match ExceptionClass2,
then Python propagates the exception up the call stack, and repeats the algorithm at each level



Catching Exceptions

There are many standard Python exception classes

For reference

[see slide]

We've seen Exception and EOFError

Catching Exceptions

```
BaseException (cont.)
   Exception (cont.)
      OSError
         BlockingIOError
         ChildProcessError
         ConnectionError
            BrokenPipeError
            ConnectionAbortedError
            ConnectionRefusedError
            ConnectionResetError
         FileExistsError
         FileNotFoundError
         InterruptedError
         IsADirectoryError
         NotADirectoryError
         PermissionError
         ProcessLookupError
         TimeoutError
```

Python standard exceptions (cont.)

22

Catching Exceptions

For reference

[see slide]

Catching Exceptions

```
BaseException (cont.)
   Exception (cont.)
      ReferenceError
      RuntimeError
         NotImplementedError
         RecursionError
      StopIteration
      StopAsyncIteration
      SyntaxError
         IndentationError
            TabError
      SystemError
      TypeError
      ValueError
         UnicodeError
            UnicodeDecodeError
            UnicodeEncodeError
            UnicodeTranslateError
```

Python standard exceptions (cont.)

23

Catching Exceptions

For reference

[see slide]

We've seen ValueError

Catching Exceptions

```
BaseException (cont.)
   Exception (cont.)
      Warning
         BytesWarning
         DeprecationWarning
         FutureWarning
         ImportWarning
         PendingDeprecationWarning
         ResourceWarning
         RuntimeWarning
         SyntaxWarning
         UnicodeWarning
         UserWarning
   GeneratorExit
   KeyboardInterrupt
   SystemExit
```

Python standard exceptions (cont.)

Catching Exceptions

For reference

[see slide]

It certainly is possible to define your own exception class, that is, some class that inherits from Exception

But it's more common to use the existing exception classes
That is, when composing a function/method that must throw some exception, the common practice is to use the existing Exception class that best fits the situation
We won't cover how to compose exception classes

Aside: Exit Status

- See **circle5.py**
  - The job:
    - Same as circle4.py, but…
    - Exits with proper status

```
$ python circle5.py
Enter the circle's radius:
5
A circle with radius 5 has diameter 10
and circumference 31.415927.
$ echo $?
0
$ python circle5.py
Enter the circle's radius:
xyz
Error: Not an integer
$ echo $?
1
$ python circle5.py
Enter the circle's radius:
Error: Missing integer
$ echo $?
1
$
```

25

Aside: Exit Status

Code notes:

Recall from COS 217…
Any process exits with a status
Convention: 0 => normal exit, non-0 => abnormal exit
Scripts that execute your program can detect its exit status and behave accordingly

In Python the default exit status is 0
To exit with some other status, call the sys.exit() function

On a Unix-like system you can issue the command “echo $?” to determine the exit status of the process that exited most recently

[see slide]

Agenda

- Data types and operators
- Terminal I/O
- Catching exceptions
- **Statements**
- Throwing exceptions

Statements

- See **euclidclient1.py**
  - The job:
    - Read two integers from stdin
    - Write their gcd and lcm to stdout

```
$ python euclidclient1.py
Enter the first integer: 8
Enter the second integer: 12
gcd: 4
lcm: 24
$
```

27

Statements

[see slide]

Code notes:

Statements:

try...except

while

return

abs() function

Defined in the __builtin__ module

Pronounced “dunder-builtin”

Need not be imported

Truncating integer division

Statements

- See **euclidclient1.py** (cont.)
  - Notes:
    - ***Unpacking assignment statement***

```
x, y = 1, 2
```

Traditional

```
temp = i%j
i = j
j = temp
```

Verbose

```
temp1 = j
temp2 = i%j
i = temp1
j = temp2
```

Python

```
i, j = j, i%j
```

28

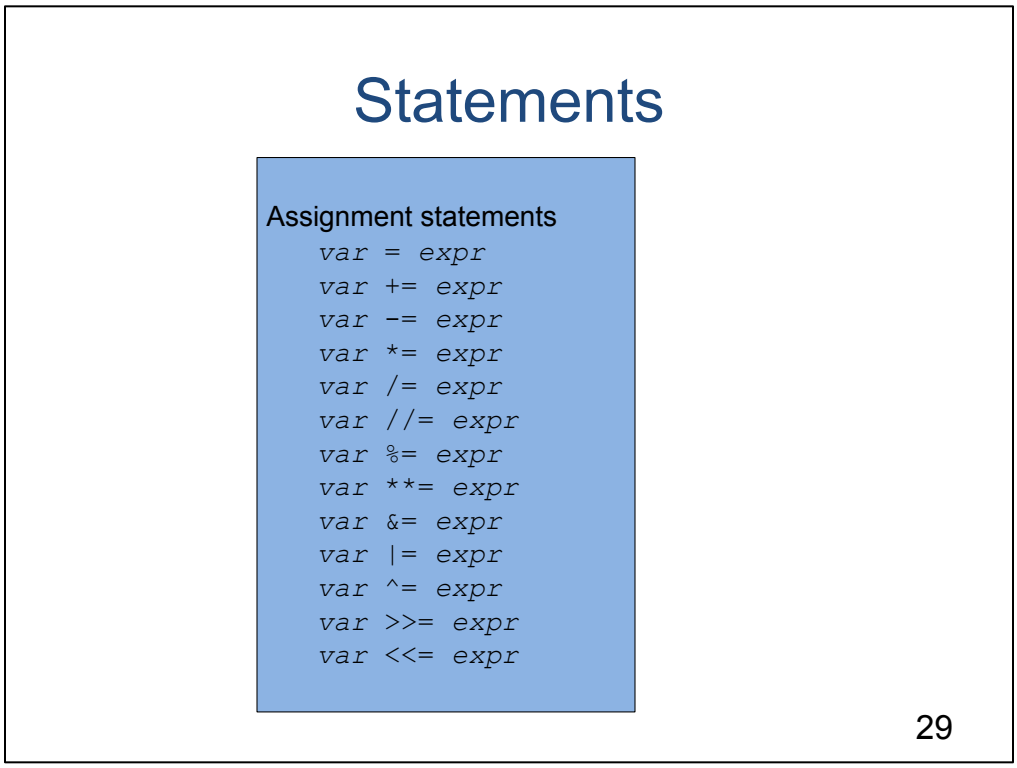
Statements

[see slide]

Code notes:

Unpacking assignment statement
Control flow statements: while, return, try

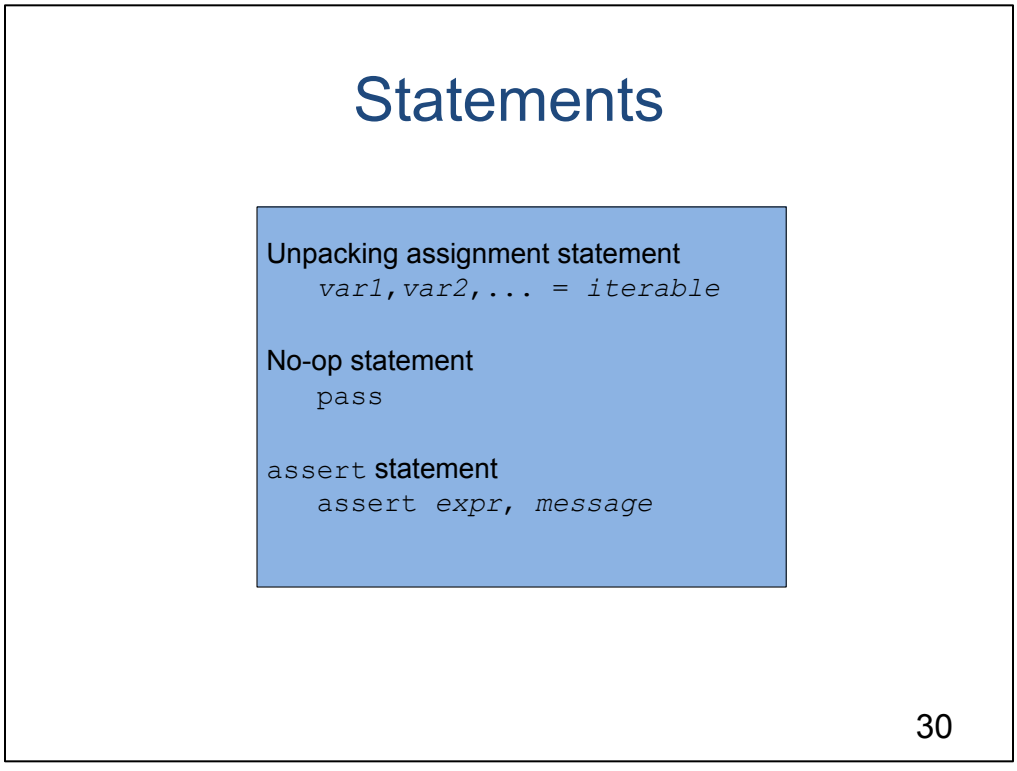
Generalizing…



Statements

For reference

[see slide]



Statements

For reference

[see slide]

Notes:

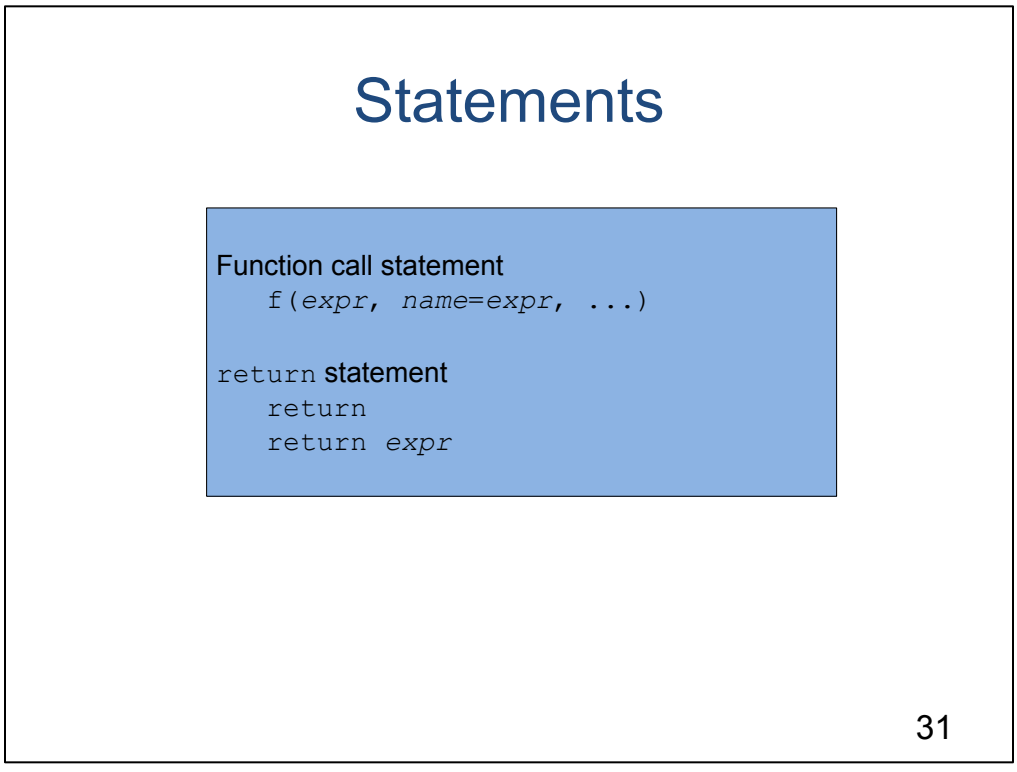
Python's pass statement is used as you would use {} in Java

assert statement

Just as C has an assert() macro, so Python has an assert statement

In C it's common to use assert() to validate parameters
In Python, as in Java, it's better to use exception handling to validate parameters

We won't use assert in Python



Statements

For reference

[see slide]

Statements

`if` statement

```
if expr:
    statement(s)
elif expr:
    statement(s)
else:
    statement(s)
```

`False`, `0`, `None`, `''`, `""`, `[]`, `()`, and `{}` indicate logical FALSE

Any other value indicates logical TRUE

32

Statements

For reference

[see slide]

Note:

The liberal representation of logical FALSE
To make C programmers happy

if i == 0: …
if not i: …

if len(somelist) == 0: …
if not somelist: …

Statements

`while` statement

```
while expr:
    statement(s)
```

`False`, `0`, `None`, `''`, `""`, `[]`, `()`, and `{}` indicate logical FALSE

Any other value indicates logical TRUE

Statements

For reference

[see slide]

Note:

The liberal representation of logical FALSE

Statements

`for` statements

```
for var in iterable:
    statement(s)
for var in range(startint, stopint):
    statement(s)
for var in range(stopint):
    statement(s)
```

`break` statement

```
break
```

`continue` statement

```
continue
```

34

Statements

For reference

[see slide]

Note:

The fundamental form of the for statement is for var in iterable
The range() function returns an iterable object
(Actually, it returns a *generator*; beyond our scope)

Statements

```
try statement
   try:
       statement(s)
   except [ExceptionType [as var]]:
       statement(s)

raise statement
   raise ExceptionType(str)
```

35

Statements

For reference

[see slide]

Notes:

Python's raise is the same as Java's throw

Agenda

- Data types and operators
- Terminal I/O
- Catching exceptions
- Statements
- **Throwing exceptions**

Throwing Exceptions

- Recall **euclidclient1.py**
  - Problem:
    - Functions fail to detect bad data

```
$ python euclidclient1.py
Enter the first integer: 0
Enter the second integer: 12
gcd: 12
lcm: 0
$ python euclidclient1.py
Enter the first integer: 0
Enter the second integer: 0
gcd: 0
Traceback (most recent call last):
  File "euclidclient1.py", line 53, in
<module>
    main()
  File "euclidclient1.py", line 42, in main
    my_lcm = lcm(i, j)
  File "euclidclient1.py", line 26, in lcm
    return (i // gcd(i, j)) * j
ZeroDivisionError: integer division or modulo
by zero
$
```

Throwing Exceptions

A function should protect itself against bad data provided by the caller
The gcd() and lcm() functions in euclidclient1.py don't do that

Specifically:

lcm() fails to detect bad data

If either int is 0, then lcm is mathematically undefined

gcd() fails to detect bad data

If both ints are 0, then gcd is mathematically undefined

[see slide]

Throwing Exceptions

- See **euclidclient2.py**
  - The job:
    - Same as euclidclient1.py except…
    - Validates parameter values

```
$ python euclidclient2.py
Enter the first integer: 8
Enter the second integer: 12
gcd: 4
lcm: 24
$ python euclidclient2.py
Enter the first integer: 0
Enter the second integer: 12
gcd: 12
lcm(i,j) is undefined if i or j is 0
$ python euclidclient2.py
Enter the first integer: 0
Enter the second integer: 0
gcd(i,j) is undefined if i and j are 0
$
```

38

Throwing Exceptions

[see slide]

Code notes:

raise statement

Throws an exception

In this case…

Instantiates an object of class ZeroDivisionError containing the given string

Throws that object

Except clause

Can reference the Exception object that was thrown

Can extract the str object that the Exception object contains

Summary

- We have covered these aspects of Python:
  - Data types and operators
  - Terminal I/O
  - Catching exceptions
  - Statements
  - Throwing exceptions