CS 11 Python track

Lecture 5: Python 3.x

Python 3.x

- Python 3.1.2 is the latest version of Python
  - (not yet available on the CS cluster)
- This lecture will summarize the main differences between Python 2.x and Python 3.x
- Comprehensive overview:

  http://docs.python.org/dev/3.0/whatsnew/3.0.html

Overview

- Python 3.x is a major revision of the language
- Several new features
- A lot of cleaning up of old features
- Many dubious old features removed entirely
  - some just deprecated; will be removed eventually
- "Deliberately incompatible" with older versions
  - so old code *will* break if run under python 3.x

This lecture

- Too many changes in Python 3.x to conveniently fit into one lecture
- Therefore, this lecture contains features that
  - are most likely to trip you up
  - are most radical
  - are most interesting (according to me)

`print` statement (1)

- The `print` statement is no longer special
  - it's just another function!
- So instead of writing

```
print "hello, world!"
```

- You write:

```
print("hello, world!")
```

`print` statement (2)

- Old:

```
print "foo", "bar", "baz"
```

- New:

```
print("foo", "bar", "baz")
```

`print` statement (3)

- Old:

```
print "foobar",   # no newline
```

- New:

```
print("foobar", end="")
print("foobar", end=" ")
    # space instead of newline
```

`print` statement (4)

- Old:

```
>>> print "There are <", 2**32, "> possibilities!"
There are < 4294967296 > possibilities!
```

- New:

```
>>> print("There are <", 2**32, "> possibilities!",
      sep="")  # no separator
There are <4294967296> possibilities!
```

`print` statement (5)

- Old:

```
>>> print  # prints a newline only
```

- New:

```
>>> print()
```

`print` statement (6)

- Old:

```
>>> print >> sys.stderr, "error!"
```

- New:

```
>>> print("error!", file=sys.stderr)
```

String formatting (1)

- Old:

```
>>> "int: %d\tfloat: %f\tstring: %s\n" %
    (10, 3.14, "this is a string")
```

- New:

```
>>> "int: {0}\tfloat: {1}\tstring: {2}\n".format(
    10, 3.14, "this is a string")
```

- (Old syntax still supported for the time being)

String formatting (2)

- Keyword syntax:

```
>>> "a: {0}\tb: {1}\tc: {c}".format(1, 2, c=3)

'a: 1    b: 2    c: 3'
```

- Literal curly braces:

```
>>> "Braces: {{}}".format()

'Braces: {}'
```

String formatting (3)

- Attributes inside curly braces:

```
>>> lst = [1, 2, 3, 4, 5]

>>> "lst[3] = {0[3]}".format(lst)

'lst[3] = 4'

>>> dict = {"foo" : 1, "bar" : 2}

>>> "dict[\"foo\"] = {0[foo]}".format(dict)

'dict["foo"] = 1'
```

String formatting (4)

- Dot syntax inside curly braces:

```
>>> import string
>>> "lowercase letters:{0.ascii_lowercase}".format(
    string)
'lowercase letters: abcdefghijklmnopqrstuvwxyz'
```

- Format specifiers:

```
>>> "pi: {0:15.8f}".format(math.pi)
'pi:       3.14159265'
```

Iterators (1)

- Iterators are a new kind of Python object
- Have been around for a while (since Python 2.2)
    - Are used more extensively in Python 3.x
- Embody the notion of "sequence that you can take the next item of"

Iterators (2)

- Iterators can be constructed from most Python sequences using the **`iter()`** function:

```
>>> i = iter(range(10))

>>> i

<listiterator object at 0x81c30>
```

Iterators (3)

- Iterators have a **`__next__()`** special method:

```
>>> i = iter(range(10))
>>> i.__next__()
0
>>> i.__next__()
1
...
```

Iterators (4)

- Continuing:

```
>>> i.__next__()
8
>>> i.__next__()
9
>>> i.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iterators (5)

- Iterating through an iterator:

```
i = iter(range(5))

for item in i:
    print(item)
```

```
0
1
2
3
4
```

Iterators (6)

- In Python 3.x, common functions like **`map()`** and **`filter()`** return iterators, not lists
- However, you can turn an iterator into a list using the **`list()`** built-in function
- Advantage of iterator: doesn't have to generate the entire list all at once

The `range()` function

- The `range()` function no longer returns a list
- Instead, it's "lazy"; it evaluates its arguments as they are needed (sort of like an iterator)
- Useful if you write *e.g.* `range(1000000)` and only use the first (say) 1000 elements
- This is also doable in Python 2.x as the `xrange()` function, which no longer exists in Python 3.x

Integer division

- Dividing two integers that don't divide evenly returns a float:

```
1 / 2      #  ==> 0.5

5 / 3      #  ==> 1.66666...
```

- If you want truncating behavior, use the new `//` operator:

```
1 // 2     #  ==> 0

5 // 3     #  ==> 1
```

Removed syntax

- Backtick syntax has been removed:

```
`foo`   # used to be same as str(foo)
```

- `<>` no longer usable for `!=`
- `from X import Y` only legal at top level (yeah!)
  - still works, but will be removed eventually

Syntax changes (1)

- `True`, `False`, `None` are reserved words
  - Good news: can't assign to them
  - Bad news: `True` still `== 1` etc. (retarded)
- Binary literals
  - Use `0b` prefix (by analogy with `0x` for hexadecimal)

```
0b11001   # ==> 25
```

- Also octal literals with `0o` prefix (who cares?)

Syntax changes (2)

- Set literals
  - `{1, "foo", 4}` is a set (not a dictionary)
  - `{}` is still the empty dictionary; use `set()` for empty sets
- Set methods:
  - `intersection`, `union`, `difference`

```
>>> {1, 2, 3}.intersection({1, 2, 5})
{1, 2}
```

- `X in <set>`

```
>>> 1 in {1, 2, 3}
True
```

Syntax changes (3)

- Set comprehensions:

```
>>> {x + y  for x in range(3)
             for y in range(3)}
{0, 1, 2, 3, 4}
```

- Note difference between this and corresponding list comprehension: no duplicates!

```
>>> [x + y  for x in range(3)
             for y in range(3)]
[0, 1, 2, 1, 2, 3, 2, 3, 4]
```

Syntax changes (4)

- Dictionary comprehensions

```
>>> {k:v for (k, v) in
     [("foo", 1), ("bar", 2), ("baz", 3)]}
{'baz': 3, 'foo': 1, 'bar': 2}
```

Syntax changes (5)

- Strings now use Unicode (16-bit) instead of ASCII (8-bit) representation for characters
  - Unicode characters can be typed in strings directly as `\uxxxx` where `x` = hexadecimal digit
- Strings preceded by the character `b` are "byte literals"; each character is one byte

```
>>> b"foo"  # byte array: [102,111,111]
b'foo'
>>> bytes([102,111,111])
b'foo'
```

Syntax changes (6)

- Byte arrays can also have characters outside of ASCII range:

```
>>> bytes([127,128,129])

b'\x7f\x80\x81'

>>> bytes([255,255,256])

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

ValueError: bytes must be in range(0, 256)
```

Syntax changes (7)

- Function argument and return value annotations

```
>>> def foo(x:int, y:int) -> int:
...     return x + y
>>> foo(3, 4)
7
>>> print(foo.__annotations__)
{'y': <class 'int'>, 'x': <class 'int'>,
'return': <class 'int'>}
```

Syntax changes (8)

- Python doesn't *do* anything with argument and return value annotations except make them available as the `__annotations__` attribute of the function

- Code introspection tools might use them for e.g. some kind of run-time type checking

- What follows the `:` or `->` doesn't have to be a type name; it can be any python expression e.g.

```
def foo(x:"this is x",
        y:"and this is y") -> 42: ...
```

Syntax changes (9)

- `nonlocal` statement
- Python allows nested functions:

```
def adder(x):
    def add(y):
        return x + y
    return add
>>> add5 = adder(5)
>>> add5(10)   # 15
```

- So far so good ...

Syntax changes (10)

- Sometimes want to modify variable in an outer scope:

```
def counter(x):
    count = x
    def increment():
        nonlocal count
        count += 1
        return count
    return increment
>>> c = counter(0)
>>> c()  # 1
```

Syntax changes (11)

- Iterable unpacking:

```
>>> a, *b, c = range(5)
>>> a
0
>>> c
4
>>> b
[1, 2, 3]
```

Built-in functions

- `raw_input()` is now just `input()`
- `reduce()` is now `functools.reduce()`
- `dict.has_key()` gone; use `x in dict`

Exceptions

- All exceptions must subclass from **`BaseException`**
- **`raise MyException(args)`** instead of **`raise MyException, args`** (allowed in Python 2.x)
- Catching: **`except MyException as e`**, not **`except MyException, e`**
- Other changes too: see the online docs for the full story

Function decorators (1)

- When you write:

```
@foo
def bar(x):
    # ... whatever ...
```

- It's the same as:

```
def bar(x):
    # ... whatever ...
bar = foo(bar)  # "decorated" function
```

Function decorators (2)

- A very common decorator:

```
class Foo:
    @staticmethod
    def bar(x):    # n.b. no "self"
        return 2 * x
>>> Foo.bar(10)
20
```

- `bar` is a method of class `Foo`, not of `Foo` instances
  - though it also works on instances

Summing up

- These changes remove language "warts", make language cleaner and more consistent
- Many more extensions/changes than I've discussed here!
  - See online docs for comprehensive list
- Will take time to absorb all the changes
- Python 3.x provides "`2to3`" program to automate most conversions