List Comprehensions

A cleaner, nicer way to do map-like operations

```python
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [2**x for x in "yogurt cheese"]
["yy", "oo", "gg", "uu", "rr", "tt", ...]
```

List Comprehensions

Syntax:

```python
>>> [e * x for x in s]
```

Equivalent to:

```python
>>> def map_fn(x): return e * x
>>> map(map_fn, s)
```

List Comprehensions

A cleaner, nicer way to do map+filter-like operations

```python
>>> [x**2 for x in range(10) if even(x)]
[0, 4, 16, 36, 64]
>>> [2**x for x in "0123456" if even(x)]
["00", "22", "44", "66"]
>>> [z[0] for z in craigslist if z[1]<3.0]
["dinosaur"]
```

List Comprehensions

Syntax:

```python
>>> [e * x for x in s if c * x]
```

Equivalent to:

```python
>>> def map_fn(x): return e * x
>>> def filter_fn(x): return c * x
>>> map(map_fn, filter(filter_fn, s))
```
**List Comprehensions**

Can “nest” the for to iterate over multiple sequences

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

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

---

**What can sequences do?**

Select
- i-th element: `s[i]`
- subsequence (“slice”): `s[i:j]`

Update -- For mutable sequences (e.g. Lists)
- Update i-th element: `s[i] = e`
- Update subsequence: `s[i:j] = e`

Member: `x in s`

Iteration: `for x in s: <body>`

map, filter, reduce

Comprehensions: `[e for x in s if c]`

---

**Quicksort in Python**

```python
def sort(L):
    if L==[]: return L
    else:
        l=sort([x for x in L[1:] if x < L[0]])
        r=sort([x for x in L[1:] if x >= L[0]])
        return(l+L[0:1]+r)
```

---

**Today: Revisit some objects**

- Exploit features and build powerful expressions

  **Base:** `int, float, complex`

  **Sequence:** `string, tuple, list`

  **Maps (Dictionary):** `key → value`

---

**Key data structure: Dictionaries**

Associative arrays, Hash tables ...

A table storing a set of “keys”,
And a “value” for each key.

Any (immutable) object can be a key!
- `int, float, string, tuples...`

Very useful!

---

**Using Dictionaries**

Unsorted list of key,value pairs

Empty Dictionary: `{}`

Non-empty Dictionary: `{k1:v1,k2:v2,...}`

Membership: `k in dict: k in d`

Lookup value of key: `d[k]`

Set value of key: `d[k]=v`
Dictionaries

```python
>>> d = {}
>>> d = dict(mexmenu)
>>> d["ceviche"] = 3.95
>>> d
{…}
>>> d["burrito"]
3.50
>>> d.keys()
...
>>> d.values()
```

```
def freq(s):
    d = {}
    for c in s:
        if c in d:
            d[c] += 1
        else:
            d[c] = 1
    return d

>>> d = plotfreq([1, 1, 3.0, "A", 3.0, "A", "A", 1, 2, 3.0, 1, "A"])
>>> d
...
>>> d = plotfreq("avrakedavra")
>>> d.keys()
...
```

You now know enough to do PA5
- Python Tutorial: How to open files, read lines
- Use the help Command
- Document every function: What does it do?

Next: What’s in a name?
More precisely:
- How should programmer think of data
- What does a variable “x” really mean?

What’s in a name?
ML (or Functional Languages)
- Name refers to a Value
- Binding maps Names to Values
- Environment list of bindings
- Environment can be extended
- Environment can’t be changed
Data model in functional PL

- Vars = names in phonebook
- Evaluation = Most recent
- Environment “frozen” in function value
  - behavior of function cannot be changed
  - easier reasoning

Data model in OO langs

- Variables “point to” objects
- Objects = boxes with data inside

Namespaces

- Manage variable names in Python
- Similar to, but different from Environments
  - Core PL concept, unifies many ideas
- We will see very important differences

Ok, but what IS a namespace ?

A mapping from names to objects

Namespaces vs. Environments

Both are maps from variables to something

Namespace          Environment

What’s the difference ?
1. Assignment
2. Updates/Mutation

1. Assignment

Basic operation in Imperative PL:

1. Compute object corresponding to e
2. Change the name “x” to refer to object
1. Assignment

Basic operation in Imperative PL:
\[ x = e \]

1. Compute object corresponding to \( e \)
2. Change the name “\( x \)” to refer to object

Assignment: changes box that name refers to

2. Update/Mutation

Change what’s inside the box (object)
- Not with immutable objects
  - eg. integers
- But with mutable objects
  - eg. arrays, lists, dictionaries

```python
>>> x = [100, 200]
>>> x
[100, 200]
>>> x[0] = "gobble gobble"
>>> x
['gobble gobble', 200]
```

How is it different from “build a new box with updated value inside”?

Aliasing

Two or more names refer to same object

“Peter Parker”
“Spider-Man”

Aliasing

Two or more names refer to same object

\[ x = [100, 200] \]
\[ y = x \]
Aliasing and Update

Two or more names refer to same object

```
>>> x = [100,200]
>>> y = x
>>> y[0] = “gobble gobble”
>>> x
['gobble gobble',200]
```

Aliasing

Two or more names refer to same object

```
>>> x = [100,200]
>>> y = x
>>> y[0] = “gobble gobble”
>>> x
['gobble gobble',200]
```

If multiple names refer to same object, update affects values of all names

Aliasing

Does not happen in Ocaml/Functional PLs
- actually it does happen (where?)
- but not exposed to the programmer

Does happen in every imperative PL
- Java, Python: names point to objects
- C: names point to memory cells

Namespaces everywhere

Namespace = map from names to objects

Notion of namespace pervades Python
- Can create namespace,
- Can name a namespace,
- Can peep inside a namespace (see what’s bound)

Go to code!

Creating Namespaces

```
a.py
x = 22
y = “this sentence is false”

>>> import a
>>> a.x
22
```

```
b.py
x = “pumpkin”
y = 3.142

>>> import b
>>> a.x
22
```
**Namespaces**

For two namespaces \(a, b\):
- names inside unrelated
- names in different spaces

\(a.x\):
attribute/name “\(x\)” in space “\(a\)”

\(b.x\):
attribute/name “\(x\)” in space “\(a\)”

Different names can point to same object!

---

**Creating Namespaces: Fun Calls**

```
x = 10
def f(y):
    y = y + x
    return y
f(x)
```

**Call-by-Value:**
- New local namespace for call
- \(y\) bound to same object (value) as \(x\)
- \(x\) binding unchanged by call

In this case, after call, local namespace disappears...

Questions:
- Why “new local namespace” (not just stack) ?
- What’s the deal with “\(x\)” not declared/bound in “\(f\)” ?
- When do we need to freeze a namespace ?

---

**Creating Namespaces: Fun Calls 2**

```
y = 0
x = [10]
def f(y):
    z = len(x)+y
    return z
f(5)
```

**Static Scoping**
Look up at runtime
Not compile time
Missing \(z\) added

---

**Creating Namespaces: Fun Calls 3**

```
>>> def g(y):
...    return y + n
...g(5)
...NameError: global name ‘n’ is not defined
```

What happened ?
Looks for “\(n\)” at run-time, when “\(g\)” is called
Can’t find “\(n\)” in local, global, builtins
Throws run-time error...
Creating Namespaces: Fun Calls 3

>>> def g(y):
>>>     return y + n
>>> g(5)
NameError...
>>> n = 10
>>> g(5)
15

What happened?
Looks for “n” at run-time, when “g” is called
Finds “n” in global, returns 15
Here “n” is a “free variable” of “g”
Needs to be “bound” in some enclosing scope

Aaargh!

Changed behavior after definition
whether or not fun works depends on what we did after fundef
Change I/O behavior too …
Unlike ML, no new binding:
just change what “n” is bound to
be careful with free variables!

Python tries to avoid “overwrites”

>>> n
100
>>> def f():
>>>     n = “smash”
>>>     print n
>>>     f()
smash
>>> n
100

Python tries to ensure you
don’t overwrite outer variables

How?
- unlike C/Java
- assignment different from reads
- no variable “declarations”
- assignment = declaration!

Python tries to avoid “overwrites”

Assignment Revisited

\[ x = e \]

1. Compute object corresponding to \( e \)
2. Change the name “x” to refer to object in the current namespace (added if missing)
What happens?

```python
>>> x = 10
>>> def g():
...   x = x + 1
...   print x
...   g()
... x
```

What happened?

```python
x = 10
def g():
  x = x + 1
  print x
```

- You may think it should print 11, and leave the global unchanged... but here is what really happens
- Since x is assigned in the function, it treats x as a local, and so it adds a binding for it.
- But then it looks up x for evaluating x+1, and it can’t find a local x, so ERROR!

What happens?

```python
>>> x = 10
>>> def g():
...   global x
...   x = x + 1
...   print x
...   g()
... x
```

What happens?

```python >>> x = [10]
>>> def g():
...   x[0] = “abc”
...   print x
...   g()
... x
```

What happens?

```python
>>> x = [10]
>>> def f(y):
...   def h(z):
...     return (y+x[0]+z)
...   return h
... foo = f(5)
>>> foo
<function object>
>>> foo(100)
105
>>> foo1 = f(-5)
>>> foo1(100)
105
```