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

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>y</td>
<td>&quot;a&quot;-&quot;b&quot;</td>
</tr>
<tr>
<td>z</td>
<td>2.9</td>
</tr>
</tbody>
</table>

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

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

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

```
X → "pumpkin"
Y → 3.142
Z → [1,2,3]
```

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

```
X → "pumpkin"
Y → 3.142
Z → [1,2,3]
```

Namespaces vs. Environments

Both are maps from variables to something

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td></td>
</tr>
<tr>
<td>Updates/Mutation</td>
<td></td>
</tr>
</tbody>
</table>

What’s the difference?

1. Assignment
2. Updates/Mutation
1. Assignment

Basic operation in Imperative PL:

\[
\begin{align*}
&x = e \\
&1. \text{Compute object corresponding to } e \\
&2. \text{Change the name “x” to refer to object}
\end{align*}
\]

Simple example

\[
\begin{align*}
i, s &= 0, 0 \\
\text{while } (i \leq 3): \\
i, s &= i + 1, s + i
\end{align*}
\]

Same name “s”
- points to different objects
- namespace is not extended

Simple example

\[
\begin{align*}
i, s &= 0, 0 \\
\text{while } (i \leq 3): \\
i, s &= i + 1, s + i
\end{align*}
\]

Same name “s”
- points to different objects
- namespace is not extended
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”

```
>>> 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
```

Aliasing

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

Aliasing

Good because ?
Avoids copying objects

Bad because ?
Sharing
Non-local effects
Memory Leaks
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"
```

```
x = "pumpkin"
y = 3.142
```

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

---

**b.py**

```
x = "pumpkin"
y = 3.142
```

>>> import a
>>> 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!

---

Namespaces

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 arg 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
- Lookup 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

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 func-definition
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”

Assignment Revisited
\[
\text{x} = e
\]

1. Compute object corresponding to e
2. Change the name “x” to refer to object in the current namespace (added if missing)
Python tries to avoid “overwrites”

```python
>>> n
100
>>> def f():
   n = "smash"
   print n
>>> f()
smash
>>> n
100
```

hence in global scope \( n = 100 \).

What happens?

```python
>>> x = 10
>>> def g0:
   x = x + 1
   print x
>>> x
10
>>> g0
>>> x
```

Remember: Assignment v. Mutation!

What happens?

```python
>>> x = [10]
>>> def g0:
   global x
   x[0] = "abc"
   print x
>>> x
[10]
>>> g0
>>> x
```

What happens?

```python
>>> x = [10]
>>> def g0:
   x[0] = "abc"
   print x
>>> x
```

Remember: Assignment v. Mutation!
What happens? (Hint: Closures!)

```python
>>> x = [10]
>>> def f(y):
...   def h(z):
...     return (y+x[0]+z)
...   return h
... >>>
>>> foo = f(5)
>>> foo
<function object>
>>> foo(100)
115
>>> foo1 = f(-5)
>>> foo1(100)
105
```