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: is k in dict: k in d

Lookup value of key: d[k]

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

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

Dictionaries

```python
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.keys()
…
>>> d = plotfreq("avrakedavra")
>>> d.keys()
…
```

You now know enough to do PA6
- Python Tutorial: How to open files, read lines
- Use the help command
- Document every function: What does it do?
What’s in a name?

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

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 refer to
- 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

1. Assignment

Basic operation in Imperative PL:

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

Namespaces vs. Environments

Both are maps from variables to something

Namespace

- bindin
- extend

What’s the difference?

1. Assignment
2. Updates/Mutation

Environment
Simple example

i, s = 0, 0
while (i <= 3):
    i, s = i+1, s+i

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. tuples, arrays, lists, dictionaries

>>> 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

\[
\begin{align*}
\text{>> } & x = [100,200] \\
\text{>> } & y = x \\
\text{>> } & y[0] = \text{“gobble gobble”} \\
\text{>> } & x
\end{align*}
\]

**Aliasing and Update**

Two or more names refer to same object

\[
\begin{align*}
\text{>> } & x = [100,200] \\
\text{>> } & y = x \\
\text{>> } & y[0] = \text{“gobble gobble”} \\
\text{>> } & x
\end{align*}
\]

**Aliasing**

Two or more names refer to same object

\[
\begin{align*}
\text{>> } & x = [100,200] \\
\text{>> } & y = x \\
\text{>> } & y[0] = \text{“gobble gobble”} \\
\text{>> } & x
\end{align*}
\]

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 (130) programmer

**Aliasing**

Does happen in every imperative PL

- Java, Python: names point to objects
- C: names point to memory cells
Aliasing

Good because?
- avoids copying

Bad because?
- sharing
- non-local effect
- 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"

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!

Creating Namespaces: Fun Calls

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

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

What happened?
Looks for “n” at run-time, when “g” is called
Can’t find “n” in local, global, builtins
Throws run-time error...

Aaargh!

Python tries to avoid “overwrites”

Change behavior after definition:
whether or not fun works depends on what we did after fundef

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”

```python
>>> 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!

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)

Python tries to avoid “overwrites”

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

>>> g()
```

What happens?

```python
>>> x
10
>>> g()
```

```python
>>> x
10
```

hence in global scope `n = 100`
What happened?

• 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 ⇒ ERROR

What happens?

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

```python
>>> x = 10
>>> def g():
    global x
    x = x + 1
    print x
>>> x = 10
>>> def g():
    global x
    x = x + 1
    print x
>>> g()
>>> x
```
What happens?

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