Announcements

• Hw6 is released, due next Monday
• Hog Contest!
  • Turn in a strategy that will be played against other students’ strategies
  • Can work in partnership (optional)
  • Win eternal 61A glory
  • See details up on the course web page!
• Trends project
  • Everything you need to complete the project will be covered by the end of this lecture
  • Recommended you find a partner
Midterm

- Midterm is Thursday, 7pm
  - Info page up: http://inst.eecs.berkeley.edu/~cs61a/su13/exams/midterm1.html
  - Staff cheat sheet is up on the mt1 page
  - Two exam rooms:
    - 2050 VLSB for logins cs61a-aa through cs61a-hz
    - 10 Evans for everyone else
  - Lists are on the midterm.
    - Need to know how to create one, how to select elements, and how to use list comprehensions
    - Mutation and assignment of lists are NOT covered
  - Objects, dictionaries, and mutable data will NOT be covered on midterm 1
Sequence arithmetic

Some Python sequences support arithmetic operations

```python
>>> city = 'Berkeley'
>>> city + ', CA'
'Berkeley, CA'

>>> "Don't repeat yourself! " * 2
"Don't repeat yourself! Don’t repeat yourself! "

>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> (1, 2, 3) + (4, 5, 6, 7)
(1, 2, 3, 4, 5, 6, 7)
```
Sequences as conventional interfaces

We can apply a function to every element in a sequence. This is called *mapping* the function over the sequence.

```python
>>> fibs = tuple(map(fib, range(8)))
>>> fibs
(0, 1, 1, 2, 3, 5, 8, 13)
```

We can extract elements that satisfy a given condition.

```python
>>> even_fibs = tuple(filter(is_even, fibs))
>>> even_fibs
(0, 2, 8)
```

We can compute the sum of all elements.

```python
>>> sum(even_fibs)
10
```

Both `map` and `filter` produce an iterable, not a sequence.
Iterables

Iterables provide access to some elements in order but do not provide length or element selection.

Python-specific construct; more general than a sequence.

Many built-in functions take iterables as argument:

- `tuple`: Construct a tuple containing the elements.
- `map`: Construct a map that results from applying the given function to each element.
- `filter`: Construct a filter with elements that satisfy the given condition.
- `sum`: Return the sum of the elements.
- `min`: Return the minimum of the elements.
- `max`: Return the maximum of the elements.

For statements also operate on iterable values.
Sequences and Iterables

• Iterables work in many built-in functions
  • for element in iterable_object: ...

• However, iterables do not necessarily have element selection or length capabilities
  • x = map(lambda num: num * 3, (5, 6, 7, 8))
  • len(x) is an error
  • x[2] is an error

• Sequences are iterables. Thus, also work in many built-in functions
  • for element in (1, 2, 3, 4, 5): ...
  • x = tuple(map(lambda num: num * 3, (5, 6, 7, 8)))
  • len(x)
  • x[2]
Generator expressions

One large expression that combines mapping and filtering to produce an iterable

\[(\text{<map exp> for <name> in <iter exp> if <filter exp>})\]

- Evaluates to an iterable.
- \(<\text{iter exp}>\) is evaluated when the generator expression is evaluated.
- Remaining expressions are evaluated when elements are accessed.

No-filter version: \[(\text{<map exp> for <name> in <iter exp>})\]

Precise evaluation rule introduced in Chapter 4.
Reducing a Sequence

Reduce is a higher-order generalization of max, min, and sum.

```python
>>> from operator import mul
>>> from functools import reduce
>>> reduce(mul, (1, 2, 3, 4, 5), 1)
120
```

First argument: A two-argument function
Second argument: an iterable object

Like accumulate from Homework 2, but with iterables

```python
def accumulate(combiner, start, n, term):
    return reduce(combiner,
                   map(term, range(1, n + 1)),
                   start)
```
More Functions on Iterables (Bonus)

Create an iterable of fixed-length sequences

```python
>>> a, b = (1, 2, 3), (4, 5, 6, 7)
>>> for x, y in zip(a, b):
...     print(x + y)
... 
... 5
... 7
... 9
```

The **itertools** module contains many useful functions for working with iterables

```python
>>> from itertools import product, combinations
>>> tuple(product(a, b[:2]))
(((1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5))
>>> tuple(combinations(a, 2))
(((1, 2), (1, 3), (2, 3))
```

Produces tuples with one element from each argument, up to length of smallest argument.
Lists

```python
>>> a = [3, 1, 2]
>>> a
[3, 1, 2]
>>> len(a)
3
>>> a[1]
1
>>> c, d = a, a[:]
>>> a, c, d
([3, 1, 2], [3, 1, 2], [3, 1, 2])
>>> c[0] = 4
>>> a, c, d
([4, 1, 2], [4, 1, 2], [3, 1, 2])
>>> d[0] = 5
>>> a, c, d
([4, 1, 2], [4, 1, 2], [5, 1, 2])
>>> a[1:2] = [7, 8, 9]
>>> a, c, d
([4, 7, 8, 9, 2], [4, 7, 8, 9, 2], [5, 1, 2])
```

Create a list using square brackets

Lists are sequences

Bind another name to a list or a slice of a list

Modify contents of a list

wut()?
Objects

An object is a representation of information.

All data in Python are objects.

But an object is not just data; it also bundles behavior together with that data.

An object’s type determines what data it stores and what behavior it provides.

```python
>>> type(4)
<class 'int'>

>>> type([4])
<class 'list'>
```
Object Attributes

All objects have attributes
We use dot notation to access an attribute

```python
>>> (4).real, (4).imag
(4, 0)
```

An attribute may be a *method*, which is a type of function, so it may be called

```python
>>> [1, 2, 1, 4].count(1)
2
```

Notice that we did not have to pass in the list as an argument; the method already knows the object on which it is operating
Creating and Distinguishing Objects

Calling the constructor of a built-in type creates a new object of that type

Objects can be distinct even if they hold the same data

The is and not is operators check if two objects are the same

```python
>>> [1, 2, 1, 4] is [1, 2, 1, 4]
False
```

Compare to `==`, which checks for equality, not sameness

```python
>>> [1, 2, 1, 4] == [1, 2, 1, 4]
True
```
Objects and Assignment

Assignment does not create a new object

1    \[a = [3, 1, 2]\]
\[\rightarrow\]
2    \[c, d = a, a[:]\]

But slicing does!

In our environment diagrams, assignment copies the arrow.
The “arrow” is called a *pointer* or *reference*.
Multiple names can *point to* or *reference* the same object.
Immutability

An object may be *immutable*, which means that its data cannot be changed.

Most of the types we have seen so far are immutable:

- ints, floats, booleans, tuples, ranges, strings

For an immutable type, it doesn’t matter whether or not two equal objects are the same:

Neither can change, so one is as good as the other.

```python
>>> e, f = 300, 300
>>> e is f
True
>>> e = 300
>>> f = 300
>>> e is f
False
```
Mutable Types

Mutable objects, on the other hand, can change, and any change affects all references to that object.

So we need to be careful with mutation.

1  a = [3, 1, 2]
2  c, d = a, a[:]
3  c[0] = 4
4  d[0] = 5
List Methods

Lists have many useful methods

- **append**: add an element to the end of a list
- **extend**: add all elements from an iterable to the end of the list
- **count**: count the number of occurrences of a value
- **pop**: remove an element from the end of a list
- **sort**: sort the elements of a list

These methods (except **count**) mutate the list

Compare to `sorted(x)`, which returns a new list

Call `dir(list)` to see a full list of attributes
List Comprehensions

We can construct a list using a *list comprehension*, which is similar to a generator expression

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

- Evaluates to a list.
- `<iter exp>` is evaluated once.
- `<name>` is bound to an element, and `<filter exp>` is evaluated. If it evaluates to a true value, then `<map exp>` is evaluated, and its value is added to the resulting list.

```
>>> [3 / x for x in range(4) if x != 0]
[3.0, 1.5, 1.0]
```
Sequences map integers to values

What if we wanted arbitrary values in the domain?

We use a dictionary

```python
>>> a = [3, 1, 2]

>>> colors = {'eric': 'blue',
            'steven': 'red',
            'mark': 'green',
            'albert': 'gold'}

>>> colors['eric']
'blue'
```
Dictionary Features

Dictionaries are not sequences, but they do have a length and are iterable

- Iterating provides each of the keys in some arbitrary order

```python
>>> for person in colors:
...    print colors[person]
...    ...
### prints colors in unspecified order
```

Dictionaries are mutable

```python
>>> colors['eric'] = 'fuchsia'
```

There are dictionary comprehensions, which are similar to list comprehensions

```python
>>> {p: colors[p] + 'ish' for p in colors}
{'steven': 'redish', 'mark': 'greenish',
'albert': 'goldish', 'eric': 'blueish'}
```
Limitations on Dictionaries

Dictionaries are unordered collections of key-value pairs.

Dictionary keys do have two restrictions:

• A key of a dictionary cannot be an object of a mutable built-in type.
• Two keys cannot be equal. There can be at most one value for a given key.

This first restriction is tied to Python's underlying implementation of dictionaries.

The second restriction is an intentional consequence of the dictionary abstraction.
A Function with Evolving Behavior

Let's model a bank account that has a balance of $100

Return value: remaining balance

>>> withdraw(25)
75

>>> withdraw(25)
50

Different return value!

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
35

>>> withdraw = make_withdraw(100)

Argument: amount to withdraw

Second withdrawal of the same amount

Where's this balance stored?

Within the function!
Persistent Local State

```python
#initialize a withdraw...

... 

>>> withdraw(25)
75

>>> withdraw(25)
50
```

A function with a parent frame

The parent contains local state

Every call changes the balance
**Reminder: Local Assignment**

**Execution rule for assignment statements:**

1. Evaluate all expressions right of =, from left to right.
2. Bind the names on the left the resulting values in the first frame of the current environment.
Non-Local Assignment

```python
def make_withdraw(balance):
    """Return a withdraw function with a starting balance."
    
def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Insufficient funds'
            balance = balance - amount
        return balance
    return withdraw
```

Declare the name "balance" nonlocal

Re-bind balance where it was bound previously
The Effect of Nonlocal Statements

nonlocal <name>, <name 2>, ...

Effect: Future assignments to that name change its pre-existing binding in the first non-local frame of the current environment in which that name is bound.

From the Python 3 language reference:

Names listed in a nonlocal statement must refer to pre-existing bindings in an enclosing scope. Names listed in a nonlocal statement must not collide with pre-existing bindings in the local scope.

http://docs.python.org/release/3.1.3/reference/simple_stmts.html#the-nonlocal-statement
http://www.python.org/dev/peps/pep-3104/
## Effects of Assignment Statements

<table>
<thead>
<tr>
<th>Status</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>• No nonlocal statement</td>
<td>Create a new binding from name &quot;x&quot; to object 2 in the first frame of the current environment.</td>
</tr>
<tr>
<td>• &quot;x&quot; is not bound locally</td>
<td></td>
</tr>
<tr>
<td>• No nonlocal statement</td>
<td>Re-bind name &quot;x&quot; to object 2 in the first frame of the current env.</td>
</tr>
<tr>
<td>• &quot;x&quot; is bound locally</td>
<td></td>
</tr>
<tr>
<td>• nonlocal x</td>
<td>Re-bind &quot;x&quot; to 2 in the first non-local frame of the current environment in which it is bound.</td>
</tr>
<tr>
<td>• &quot;x&quot; is bound in a non-local frame</td>
<td></td>
</tr>
<tr>
<td>• nonlocal x</td>
<td>SyntaxError: no binding for nonlocal 'x' found</td>
</tr>
<tr>
<td>• &quot;x&quot; is not bound in a non-local frame</td>
<td></td>
</tr>
<tr>
<td>• nonlocal x</td>
<td>SyntaxError: name 'x' is parameter and nonlocal</td>
</tr>
<tr>
<td>• &quot;x&quot; is bound in a non-local frame</td>
<td></td>
</tr>
<tr>
<td>• &quot;x&quot; also bound locally</td>
<td>x = 2</td>
</tr>
</tbody>
</table>
Python Particulars

Python pre-computes which frame contains each name before executing the body of a function.

Therefore, within the body of a function, all instances of a name must refer to the same frame.

```python
def make_withdraw(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw

wd = make_withdraw(20)
wd(5)
```

UnboundLocalError: local variable 'balance' referenced before assignment