CIS 210 Introduction to Computer Science Winter 2017

- Objects/vars/assignment, revisited
- Lists
- Dictionaries

```
Computers are good at following instructions, but not at reading your mind.
~ Donald Knuth
```

CIS 210 / Winter 2017

**CIS 210 Learning Outcomes**

- understand, develop, implement, and algorithms for computational problem solving;
- use structured design and testing methods to develop and implement programs;
- read, write, revise, document, test, and debug code;
- demonstrate robust mental models of data representation and code execution;
- demonstrate good understanding of a high level programming language;
- introduce and/or implement a sampling of classic computer science problem domains and algorithms.

---

CIS 210 Introduction Computer Science Winter 2017

Checking understanding

**What can we infer about f1, f2, f3?**

```python
def foo():
    
    res = f1()
    if f2():
        f(3)

    return res
```

---

CIS 210 Introduction Computer Science Winter 2017

Checking understanding

**What can we infer about f1?**

```python
def foo():
    
    res = f1()
    if f2():
        f(3)

    return res
```

f1 returns a value
f2 returns Boolean value
f3 returns None (side effect)
Bug Collection

def foo():
    '''
    return 99
    '''
    f1()
    if f2():
        f(3)
    return res

def f1():
    return 99

res = f1()
if f2():
    f(3)
return res

Python objects

Recall:
value(s)
type – range of values and operations
id – memory location

Python objects are strongly typed

>>> 210     >>> ‘210’
210     ‘210’
>>> type(210)     >>> type(‘210’)
<class ‘int’>     <class ‘str’>
>>> len(210) ×     >>> len(‘210’)
>>> 210 / 3     >>> ‘210’ / 3 ×
>>> 210 + 3     >>> ‘210’ + ‘3’
>>> 210.0 + 3     >>> ‘210’ + 3 ×
Python objects are *dynamically typed*

```python
a = 10
a = 'hello, world'
b = a
c = False
```

Assignment statements

**static typing**

```python
var a : int
a = 4
a = 'hello'  # X
b = 4  # X
```

Assignment statements

**dynamic typing**

```python
ctr = 0
for i in range(10):
    if foo(i):
        ctr = ctr + 1
    else:
        ctr = 'my error message'
result = ctr / 10
```

Assignment statements

```python
ctr = 0
for i in range(10):
    if foo(i):
        ctr = ctr + 1
    else:
        ctr = ctr + 2
print(ctr)
```
Variable assignment, revisited

```python
>>> a = 10
>>> b = 20
>>> a = b
>>> a
??
```  

For example,

```python
>>> x = 10
>>> x
10
>>> type(x)
<class 'int'>
>>> id(x)
4297261440
```  

```python
>>> x = 20
>>> x
20
>>> type(x)
<class 'int'>
>>> id(x)
4297261760
```

A variable is a named reference (pointer) to a data object.

When the variable is given a new value, the reference (pointer) is updated (not the value).
For example,

```python
>>> x = 20
>>> x
20
>>> id(x)
4297261760
>>> x = 30
>>> id(x)
4297262080
```

```
>>> b = 20
>>> a = b
>>> a
20
>>> b = 30
>>> a
??
```

Assignment statements

```python
>>> a = 10
>>> c = a
>>> a = a * 10

>>> c
??
```
Python collections

Sequential

(Strings,) Tuples, Lists

Unordered

Dictionaries, Sets, Frozensets

Python collections* - Sequential

<table>
<thead>
<tr>
<th>Strings*</th>
<th>Tuples</th>
<th>Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered</td>
<td>ordered</td>
<td>ordered</td>
</tr>
<tr>
<td>sequential ops</td>
<td>sequential ops</td>
<td>sequential ops</td>
</tr>
<tr>
<td>characters</td>
<td>multiple types</td>
<td>multiple types</td>
</tr>
<tr>
<td>(including lists)</td>
<td>(including lists)</td>
<td></td>
</tr>
<tr>
<td>immutable</td>
<td>immutable</td>
<td>mutable</td>
</tr>
</tbody>
</table>

Python collections – Sequential

<table>
<thead>
<tr>
<th>Strings</th>
<th>Tuples</th>
<th>Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 'abc'</td>
<td>y = ('a', 'b', 'c')</td>
<td>z = ['a', 'b', 'c']</td>
</tr>
<tr>
<td>x[0]</td>
<td>y[1]</td>
<td>z[2]</td>
</tr>
<tr>
<td>'a' in x</td>
<td>'b' in y</td>
<td>'0' in z</td>
</tr>
<tr>
<td>x = 'xyz'</td>
<td>y = ('a', 1, True)</td>
<td>z = ['a', 1, (2,3)]</td>
</tr>
</tbody>
</table>

Particularly important for understanding mutable data types:

- assignment and expressions
- variables are names and references (pointers) to memory locations
- returned values (incl. None) and side effects
### Updating an integer value:

```python
>>> anum = 44
>>> anum + 55
??
>>> anum
??
>>> anum = anum + 55
>>> anum
??
```

### Updating a string:

```python
>>> astr = 'abc'
>>> astr.upper()
'ABC'
>>> astr

>>> astr = astr.upper()
>>> astr
'ABC'
```
>>> x = 'xyz'
    >>> x[0] = 'z'  X
    ?? how do we get to x = 'zyz'

>>> x = 'zyz'  \text{-or-}
    >>> x = 'z' + x[1:]  

\text{Lists are a mutable data type  
(and strings and tuples are not)}

>>> y = ['a', True, 100]  
    >>> y[0] = 'b'
Lists are a mutable data type
(and strings and tuples are not)

>>> y = ['a', True, 100]
>>> y[0] = 'b'

>>> y = ['a', True, 100]
>>> y[0] = [[1], [2,3]]

Can change the overall size of the list
during function execution

>>> y.append(99)
>>> y
[1, 2, 3, 99]

>>> y.insert(3, 98)
>>> y
[1, 2, 3, 98, 99]

>>> y.remove(2)
>>> y
[1, 3, 98, 99]

Lists are a mutable data type

can change the value of a complex object (including size) during program execution
  flexible; powerful; convenient; clear

also
  expensive (memory management)

→ Python updates the object IN PLACE
Lists are mutable. They can be updated in place.

Many list methods update a list as a side effect – and return None.

```python
>>> y = [1, 2, 3]
>>> id(y)
4353070720

>>> y = [99] + y[1:]
>>> id(y)
4353072088
```

```python
>>> y = [1, 2, 3]
>>> id(y)
4331561040

Lists are mutable. They can be updated in place.

>>> y[0] = 99
>>> y
[99, 2, 3]
>>> id(y)
4331561040
```

```python
>>> y.append(99)
>>> y
[1, 2, 3, 99]

>>> y = y.append(100)
>>> y
??
```
Many list methods update a list as a side effect – and return None

```python
>>> mystr = 'bye'
>>> myl = [1, False, 'hi']
>>> mystr[0] = 'd'
>>> myl[0] = 99
>>> mystr = mystr.upper()
>>> myl = myl.reverse()
>>> mystr = 'hello'
>>> myl = [10, 12, 2]
>>> yourstr = mystr
>>> yourl = myl
>>> yourstr = yourstr.upper()
>>> yourl = yourl.reverse()
>>> yourstr
'HELLO'
>>> yourl
[2, 12, 10]
```

Aliasing is also an issue

```python
>>> yourstr = mystr
>>> yourl = myl
>>> yourstr = yourstr.capitalize()
>>> yourl = yourl.reverse()
>>> yourstr
'Hello'
>>> yourl
[2, 12, 10]
```

Recall:

```python
>>> b = 20
>>> id(b)
4297261760
>>> a = b
>>> id(a)
4297261760
# a is an alias for b
>>> b = 30
>>> id(b)
4297261760
# same reference, 2 names
>>> a = 20
>>> id(a)
4297262080
# b's reference updated when b is
>>> id(b)
4297261760
# a is the old reference (val=20)
```
```python
>>> y = [1, 2, 3]
>>> id(y)
4331561040

>>> x = y  # x is an alias for y
>>> id(x)
4331561040

>>> y[1] = 99
>>> id(y)  # same reference, 2 names
4331561040

>>> x  # -> x is still an alias of y
??
```

```python
>>> alist = [1, 2, 3]
>>> id(alist)
4328102872

>>> alist[1] = 99
>>> id(alist)  # this stays the same
4328102872

>>> alist[1] = 99
>>> id(alist)  # this has changed
4297264288
```

```python
>>> alist = [1, 2, 3]
>>> id(alist)
4328102872

>>> alist[1] = 99
>>> id(alist)
4297261184

>>> alist  # -> [0] + alist
??
```

```python
>>> alist = [1, 99, 3]
>>> id(alist)
4328102872

>>> alist[1] = 99
>>> id(alist)
4297264288

>>> alist  # -> [0] + alist + [1, 99, 3]
??
```

```python
>>> alist = [0] + [1, 99, 3]
>>> id(alist)
4351899768

>>> alist.append(101)
>>> id(alist)
4351899768

>>> alist  # -> [0] + [1, 99, 3, 101]
??
```
Lists are a \textbf{mutable} data type \hspace{1em} \# powerful, convenient

```python
>>> myl = [True, 'Oregon', 99]
>>> id(myl)
4359098952

\textbf{content can be changed after object is created}

```python
>>> myl.append([1, 2]) \hspace{1em} \# side effect
>>> myl
[True, 'Oregon', 99, [1, 2]]
```

\textbf{content is changed in place}

```python
>>> id(myl) \hspace{1em} \# aliases are also changed
4359098952
```

```python
>>> y = [1, 2, 3]
>>> id(y)
4331561040

>>> x = y
>>> x = y.copy \hspace{1em} \# or y[:] or list(y)
>>> id(x)
4331561040
>>> id(x)
4331561040

>>> x[0] = 99
>>> x[0] = 99
>>> id(y)
4331561040
>>> id(y)
4331561040

```python
>>> li = [1, 2, [3, 4], [5, 6]]
>>> import copy
>>> li2 = li.copy()
```python
>>> li[0] = 7
>>> li[3][0] = 99
>>> li
[7, 2, [3, 4], [5, 6]]
>>> li2
[1, 2, [3, 4], [5, 6]]
>>> li[2][0] = 8
>>> li3 = copy.deepcopy(li)
>>> li3
[7, 2, [3, 4], [5, 6]]
>>> li
[7, 2, [8, 4], [5, 6]]
>>> li2
[1, 2, [8, 4], [99, 6]]
>>> li3
[7, 2, [8, 4], [5, 6]]
```
RECALL: PARAMETER PASSING IS CALL BY ASSIGNMENT → ALIASING

def bar(x):
    y = foo(x)
    print('bar-x:', x, 'bar-y:', y)
    return None

def foo(x):
    y = x.pop()
    return y

x = ['CIS 210', 'CIS 211', 'CIS 212']
bar(x)
z = x.copy()
bar(z)
print('global-z:', z)
print('global-x:', x)

Python collections – Tuples, Lists

✓ Sequential (like strings), heterogeneous collection of references to objects

Tuples (like strings) are an immutable data type:
content cannot be changed after it is created

Lists are a mutable data type
content can be changed after it is created

Lists (and dictionaries and sets) are mutable data types
content can be changed after it is created
content is changed in place
content of any alias is also changed
parameter passing creates an alias
id function can help us see this copy object to avoid aliasing

Python collections – Sequential

Lists – heterogeneous, mutable – are a very flexible and powerful data type:
Use wisely!

Is a list the best choice for representing data?

Does the data need to be changed?

No → tuple – safer and faster
Yes → list
## Python collections - **Dictionary**

<table>
<thead>
<tr>
<th></th>
<th>Lists</th>
<th>Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered</td>
<td>ordered</td>
<td>ordered</td>
</tr>
<tr>
<td>key access</td>
<td>sequential ops</td>
<td>sequential ops</td>
</tr>
<tr>
<td>multiple types</td>
<td>multiple types</td>
<td>multiple types</td>
</tr>
<tr>
<td>(keys immutable)</td>
<td>(including lists)</td>
<td>(including lists)</td>
</tr>
<tr>
<td>mutable</td>
<td>mutable</td>
<td>immutable</td>
</tr>
</tbody>
</table>

```python
>>> binaryD = {}
>>> binaryD[0] = '00000000'
>>> binaryD[1] = '00000001'
>>> binaryD[2] = '00000010'
>>> binaryD[3] = '00000011'
>>> binaryD
{0: '00000000', 2: '00000010', 3: '00000011', 1: '00000001'}
>>> binaryD[1]
'00000001'
```

```python
>>> binaryD.values()
dict_values(['00000000', '00000001', '00000010', '00000011'])
>>> list(binaryD.values())
['00000000', '00000001', '00000010', '00000011']
```

```python
>>> for item in binaryD.values():
...     print(item)
...00000000
...00000011
...00000001
...00000010
```

```python
>>> binaryD.keys()
dict_keys([2, 0, 3, 1])
>>> list(binaryD.keys())
[2, 0, 3, 1]
```

```python
>>> list(binaryD.items())
[(0, '00000000'), (1, '00000001'), (2, '00000010'), (3, '00000011')]
```