The only way to learn is by practice (with advice). That’s how you learn to ride a bike and that’s how you learn to do anything. -- Roger Schank

Data

An assortment of items, often numerical, that have been observed, measured, or collected by some means, that represent the starting point for analysis that can be done in an attempt to understand the data and understand underlying characteristics that may be present. (text)

Python collections

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Unordered</th>
<th>Immutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strings, Tuples, Lists</td>
<td>Dictionaries, Sets, Frozensets</td>
<td>Strings, Tuples, Frozensets</td>
</tr>
<tr>
<td>Immutable</td>
<td>Tuples, Frozensets</td>
<td>Tuples, Lists</td>
</tr>
<tr>
<td>Strings, Tuples, Lists</td>
<td>Dictionaries, Sets, Frozensets</td>
<td>Strings, Tuples, Frozensets</td>
</tr>
</tbody>
</table>

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>multiple types (including lists)</td>
<td>multiple types (including lists)</td>
<td></td>
</tr>
<tr>
<td>immutable</td>
<td>immutable</td>
<td>mutable</td>
</tr>
</tbody>
</table>

<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>
Recall: Updating a string:

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

```python
>>> x = 'xyz'
>>> x[0] = 'z'
✗
>>> x = 'z' + x[1:]
>>> x = '
zyz'
>>> x
'zyz'
```

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

Lists are a mutable data type (and strings and tuples are not)

```python
>>> y = ['a', True, 100]
>>> y[0] = [[1], [2,3]]
>>> y
[[[1], [2,3]], True, 100]
>>> y.append(101)
[[[1], [2,3]], True, 100, 101]
```
Lists are a mutable data type

can change the value of a complex object (including size) during program execution

flexible; powerful; convenient

also

expensive (memory management)

Python updates the object IN PLACE

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

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

>>> y.append(100)
>>> y
[99, 3, 100]
>>> id(y)
4390703048

Can change the size of the list

>>> y.append(100)
>>> y
[99, 2, 3, 100]
>>> id(y)
4331561040

>>> y.remove(2)
>>> y
[99, 3, 100]
>>> id(y)
4331561040

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

```python
>>> mystr = 'bye'
>>> mystr[0] = 'r'
>>> mystr = mystr.upper()
>>> mystr = 'hello'
>>> yourstr = mystr
>>> mystr; yourstr
>>> myl = [1, False, 'hi']
>>> myl[0] = 99
>>> myl.reverse()
```
CIS 210

>>> y = [1, 2, 3]
>>> id(y)
4319729712

>>> x = y
>>> x = y.copy # or y[:], or list(y)

>>> id(x)
4319729712

>>> y[0] = 99
>>> x
[99, 2, 3]

CIS 210

Important background for understanding mutable data types:

• what happens during variable assignment
  – variables are names and references (pointers) to memory locations where a value (object) is stored
  – two variable (names) may reference the same object (value) - aliasing

• what happens when a function is executed
  – activation record on function call stack; local namespace
  – parameter passing by assignment – more aliasing
  – function execution may result in side effects - persist after the function is done executing (e.g., print, update mutable type)
  – at return keyword (or when end of the code is reached):
    • activation record is deleted
    • function returns a value (possibly None)
    • Python resumes processing where the function was called

CIS 210

Nested lists

>>> myl
[True, 'Oregon', 99, [1, 2]]

>>> myl[0]
True

>>> myl[1]
'Oregon'

>>> myl[2]
99

>>> myl[3]
[1, 2]

CIS 210

>>> li1 = [1, 2, [3, 4], [5, 6]]
>>> li2 = li1.copy()
>>> li1[0] = 'hi'

>>> li1
['hi', 2, [3, 4], [5, 6]]

>>> li2
[1, 2, [3, 4], [5, 6]]
def bar(x):
    def foo(x):
        '''
        '''
        y = foo(x)
        y = x.pop()
        return y
    return None
x = ['CIS 210', 'CIS 211', 'CIS 212']
bar(x)
z = x.copy()
bar(z)
print('global-z:', z)
print('global-x:', x)

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 – **Dictionary**

<table>
<thead>
<tr>
<th>Dictionary</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 collections – Sequential

Lists – heterogenous, 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 – 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

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

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

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

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

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

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

CIS 210

Roman numerals:

```python
roman = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'M': 1000}
```

```python
roman['X']
```

Python collections / mutable data types

- Testing and debugging, cont'd
- Midterm review

“Good [programming] comes from experience, and a lot of that comes from bad [programming].”

– Will Rogers (sort of)

Recall:

Goal: reliable program that runs and results in correct output according to problem specification

Programming Errors:
- syntax
- runtime
- logical (semantic)
- documentation (specification, basic examples/tests)

Goal: reliable program that runs and results in correct output according to problem specification

Software engineering best practices:

- Style guidelines support development of reliable, reusable code
- Designing tests that can detect programming errors is an integral part of writing reliable code
Goal: reliable program that runs and results in correct output according to problem specification

Software engineering best practices:

→ systematic approach to formulating tests:
  • simple/basic examples
  • edge (boundary) conditions
  • for different types of expected input
  • for different types of expected results
  • white box tests supplement black box tests

→ systematic approach to testing:
  • unit tests
  • integrated tests
  • regression tests

That's a lot of testing!

Test early, test often → automate testing to make it practical.

Automated testing:

• supports comprehensive testing
• communicates comprehensive testing
• supports regression testing

So far – doctest.testmod() great as far as it goes, but for comprehensive testing

• docstrings too long
• no control over error report
• same tests may be useful for more than one system

Automating testing:

• supports comprehensive testing
• communicates comprehensive testing
• supports regression testing

def ctemp_to_ftemp(ctemp):
   """(number) -> float
   equivalent to
   ftemp = ctemp * 9/5 + 32
   return ftemp"
   return (ctemp_to_ftemp(100) == 212.0 and
           ctemp_to_ftemp(0) == 32.0 and
           ctemp_to_ftemp(30) == 86.0 and
           ctemp_to_ftemp(21.1) == 69.98)

def test_ctemp_to_ftemp():
   """() -> boolean """
   return (ctemp_to_ftemp(100) == 212.0 and
           ctemp_to_ftemp(0) == 32.0 and
           ctemp_to_ftemp(30) == 86.0 and
           ctemp_to_ftemp(21.1) == 69.98)
   }

def test_ctemp_to_ftemp():
   """() -> boolean """
   if ctemp_to_ftemp(100) != 212.0:
     return False
   if ctemp_to_ftemp(0) != 32.0:
     return False
   if ctemp_to_ftemp(30) != 86.0:
     return False
   if ctemp_to_ftemp(21.1) != 69.98:
     return False
   return True
```python
from Testing to Debugging – Finding and Fixing Bugs

The key to all debugging is knowing what your code is supposed to do.

Understand problem specification, algorithm(s), use style guidelines and other best design practices to support development of reliable and reusable code, systematic, automated approach to testing

```
from dataset:

```python
count = 0
total = 0
for value in dataset:
    if value != '0':
        total += int(value)
        count += 1
avg = total / count
return avg
```

From Testing to Debugging – Finding and Fixing Bugs

(Novice) programming ➔ better way

Neglect to track closely what programs do ➔ know what output you are expecting

Disengage from the task when trouble occurs ➔ expect bugs; leave time for debugging

Debugging – finding and fixing bugs

Concentrate on finding why the program is doing what it is doing (not why it isn’t doing what you want it to).

- Look at the code.
- Review/hand trace the code with a colleague, friend, pet, ...
- Try bits of code in the Shell.
- Isolate the bug (use print statements to find out where the program goes wrong).
- Split the code in half.
- Change one thing at a time, for a reason.

```python
count = 0
total = 0
for value in dataset:
    if value != '0':
        total += int(value)
        count += 1
    # increment count inside the if stmt
avg = total / count
return avg
```
```python
CIS 210

count = 0
total = 0
for value in dataset:
    if value != '0':
        total += int(value)
count += 1       # increment count inside the if stmt
if count != 0:
    # guard against division by zero
    avg = total / count
else:
    return 0       # return 0 for no data at all

CIS 210

Programming/Computer Science Concepts

Computational Problem Solving: designing, implementing, checking, revising algorithms/programs.

Good programming style: function docstrings (type contract; description including parameters, returned value, and side effects if any; examples of function use), well-named variables, use of whitespace between operators and sections of code, judicious use of inline comments (why not what).

Python is a programming language and Python is an interpreter (program)
Python Shell is a REPL (read-evaluate-print loop)
Python primitive elements: Objects - value/attributes, type, memory location (id)
Combining primitive elements: Expressions - expressions evaluate to a value; overloaded operators; methods/functions/operators; short-circuit evaluation of boolean expressions
Naming values: Variables/assignment - assignment statements are not expressions and do not return a value; namespaces - builtins and global (__main__); scope; dynamic typing; reference semantics.
Other language considerations - strong typing, immutable and mutable data types
Functions are an executable data type; what happens when a function is called:
Call-by-assignment parameter passing
Call-by-value parameter passing
Functions sometimes have side effects

CIS 210 Learning Outcomes

• understand, develop, implement 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.