Python collections / mutable data types

• Testing and debugging

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.

Data Structures
Data structures are how we store and access data in a computer program. A data structure organizes the data and supports basic operations on the data (e.g., add, update, retrieve, delete).

Python collections – strings, tuples, lists, dictionaries

Python collections

Sequential
Strings, Tuples, Lists

Unordered
Dictionaries, Sets, Frozensets

Immutable
Strings, Tuples, Frozensets

Mutable
Lists, Dictionaries, Sets

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

Strings

x = 'abc'
y = ('a', 'b', 'c')
z = ['a', 'b', 'c']
x[0] y[1] z[2]

Tuples

‘a’ in x ‘b’ in y ‘0’ in z

Lists

x = 'xyz' y = ('a', 1, True) z = ['a', 1, (2, 3)]

x[0] = 'z' x y[1] = 99 x z[2] = False
Important 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 object)
  - 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

Recall: Updating a string:

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

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

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

**COMPARE:**

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

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

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

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

flexible; powerful; convenient
also
expensive (memory management)

Python updates the object IN PLACE

```python
>>> y = [1, 2, 3]
>>> id(y)
4331561040
>>> y[0] = 99
>>> y
[99, 2, 3]
>>> id(y)
4331561040
```

Lists are mutable.
They can be updated in place.

Can change the size of the list

```python
>>> y.append(100)
>>> y
[99, 2, 3, 100]
>>> id(y)
4331561040
>>> y.remove(2)
>>> y
[99, 3, 100]
>>> id(y)
4331561040
>>> y = [99, 3, 100]
>>> id(y)
4331561040
>>> y
[99, 3, 100]
```
Can change the size of the list

```python
>>> y.append(100)
>>> y
[99, 2, 3, 100]
>>> id(y)
4331561040
>>> y.remove(2)
>>> y
[99, 3, 100]
>>> id(y)
4331561040
>>> y = [99, 3, 100]
>>> y
[99, 3, 100]
>>> y.append(100)
>>> y
[99, 3, 100, 100]
>>> y = y.append(100)
>>> y
[99, 3, 100, 100]
>>> y
[99, 3, 100, 100]
```

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

```python
>>> x = [100, 101, 102, 103, 101]
>>> del x[2]
>>> w = x.remove(101)
>>> y = x.pop(2)
>>> z = x.pop()
>>> a = x.append(99)
>>> print(x, w, y, z, a)
```

Recall: And for list:

```python
>>> b = 20
>>> a = b
>>> b = 30
>>> b >>> a
30 20
```
Recall:
Now for list updated in place:
```python
>>> b = 20
>>> y = [1, 2, 3]
>>> a = b
>>> x = y
>>> b = 30
>>> y[1] = 99

>>> b
>>> a
>>> y

30  20  ??  ??
```

Aliasing is also an issue
```python
>>> yourstr = mystr
>>> yourl = myl
>>> mystr = mystr.capitalize()
>>> myl.reverse()

>>> mystr
>>> myl

>>> yourstr
>>> yourl

>>> yourstr
>>> yourl

>>> mystr
>>> myl
??  ??

Lists are a mutable data type  # powerful, convenient
```python
>>> myl = [True, 'Oregon', 99]
>>> id(myl)
4359098952

content can be changed after object is created
```python
>>> myl.append([1, 2])  # list updated as a side effect
>>> myl

>>> myl.reverse()  # of append method
>>> myl

content is changed in place
>>> id(myl)
4359098952

>>> x
>>> id(x)
4359639552
```

RECALL: PARAMETER PASSING IS CALL BY ASSIGNMENT  ALIASING
```python
>>> y = [1, 2, 3]
>>> id(y)
4331561040

>>> x = y
>>> id(x)
4331561040

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

>>> x
>>> id(x)
[99, 2, 3]

>>> x is y
>>> x is y
True  False
```
Important 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 object)
  - 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

**Nested lists**

```python
>>> myl
[True, ‘Oregon’, 99, [1, 2]]
>>> myl[0]
??
>>> myl[3]
??
>>> myl[3][1]
??
```

**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 – 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

**Nifty Python**

```python
>>> 100,000
??
>>> x = 10
>>> y = 20
>>> x, y = y, x
```
CIS 210

- Python collections – dictionaries
- Testing and debugging

Storing items with labels, e.g., binary numbers with their decimal equivalents, or a sales total with the day or week or month it corresponds to.

<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>

```
>>> binaryD = {}
>>> binaryD[0] = '00000000'
>>> binaryD[1] = '00000001'
>>> binaryD[2] = '00000010'
>>> binaryD[3] = '00000011'
>>> binaryD
{0: '00000000', 2: '00000010', 3: '00000011', 1: '00000001'}
>>> binaryD[2]
'00000010'
```
CIS 210

>>> roman = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'M': 1000}
>>> roman['X']

CIS 210

Python collections / mutable data types
• Testing and debugging

CIS 210

GOAL: HIGH QUALITY COMPUTER PROGRAMS –
REUSABLE/MAINTAINABLE
RELIABLE
(EFFICIENT - TIME, SPACE)
ARE WE MEETING THIS GOAL?

CIS 210

GOAL: HIGH QUALITY COMPUTER PROGRAM –
REUSABLE/RELIABLE/EFFICIENT
ARE WE MEETING THIS GOAL?

→ style guidelines support development of reusable and reliable code

CIS 210

def testsNeeded(s):
    # DOES THIS CODE DO WHAT IT IS SUPPOSED TO DO?
    '''(str) -> int'''
    if len(s) != 0:
        prev_char = s[0]
        dup_ct = 1
        high_ct = 1
        else:
            high_ct = 0
        for i in range(1, len(s)):
            if s[i] == prev_char:  # >>> testsNeeded('abbccdf')
                dup_ct += 1
            else:
                prev_char = s[i]
                if dup_ct > high_ct:
                    high_ct = dup_ct
                high_ct = dup_ct
        return high_ct

CIS 210

def testsNeeded(s):
    # DOES THIS CODE DO WHAT IT IS SUPPOSED TO DO?
    '''(str) -> int'''
    if len(s) != 0:
        prev_char = s[0]
        dup_ct = 1
        high_ct = 1
        else:
            high_ct = 0
        for i in range(1, len(s)):
            if s[i] == prev_char:  # >>> testsNeeded('abbccdf')
                dup_ct += 1
            else:
                prev_char = s[i]
                if dup_ct > high_ct:
                    high Ct = dup Ct
                high Ct = dup Ct
        return high Ct
def testsNeeded(s):
    # DOES THIS CODE DO WHAT
    """(str) -> int # IT IS SUPPOSED TO DO?
    Returns length of longest
    single-char string in s.
    # program specification

>>> testsNeeded('abccndef')
3
>>> testsNeeded('')
0
>>> testsNeeded('abcdef')
1

GOAL: HIGH QUALITY COMPUTER PROGRAM –

RELIABLE: program runs; produces correct output
            according to the specification

• program runs
• program results are correct
• program handles the unexpected gracefully
• program runs under extreme conditions
• ... meets other specifications, e.g., HCI, platforms

Goal: program runs; produces correct output
      according to problem specification

When programs DON’T run ...

A survey of programming errors ("bugs")

When programs are not reliable ...

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

TYPES OF PROGRAMMING ERRORS

When programs DON’T run ...

• syntax - program language keywords, grammar
• runtime - TypeError, NameError, IndexError, etc.

When programs run ...

but DON’T produce correct output
CIS 210
Goal: program that runs and results in correct output according to problem specification

TYPES OF PROGRAMMING ERRORS

- syntax - program language keywords, grammar
- runtime - TypeError, NameError, IndexError, etc.
- logical (semantic) – e.g., dynamic typing error, longest char string error, etc.

How can we detect logical errors?

CIS 210
GOAL: HIGH QUALITY COMPUTER PROGRAM – REUSABLE/RELIABLE/EFFICIENT

ARE WE MEETING THIS GOAL?

→ style guidelines support development of reliable, reusable code
→ designing tests that can detect programming errors is an integral part of writing reliable code

Testing starts at program design

docstring:
  type contract
  brief description
  basic examples of use

that reflect the project specification

automated testing (e.g., doctest.testmod)

CIS 210
doctest.testmod() –

program runs –
  -- no syntax errors
  -- no obvious runtime errors
  -- correct results for basic examples of use

a very good start

but it is not enough

Systematic approach to testing

- The aim of testing is to increase confidence in the software’s reliability and expose faults, so choose test cases that are likely to thoroughly check reliability and (therefore) expose as many faults as possible.

- To test a software component, we run it with selected test cases, and compare the actual outputs with the predicted outputs. Any discrepancy signifies a fault.
A systematic approach to formulating testing goals:

Simple/Basic examples:

For different types of expected input:

For different types of expected output:

Edge (boundary) conditions:

For different types of expected input:

For different types of expected output:

A note on edge (boundary) values

Edge values are common source of bugs — e.g.,

"if n < 10;" instead of "if n <= 10;"

• The possible values of a particular input or output might fall into one or more ranges.

• If so, the testing goals should include the boundary values of each range. (e.g., n = 9, 10, 11)

• And: 0, 1, empty sequence, sequence length 1, ...

A note on different types of input and output (equivalence classes)

• The possible values of a particular input or output might fall naturally into equivalence classes, such that all the values in an equivalence class should be treated uniformly.

  – For example, all strings length 1, strings with longest repeating sequence at the beginning, strings with longest repeating sequence of length 1, ...

• If so, the testing goals should include at least one or two values of each equivalence class.

• Thinking about different equivalence classes is a good way of organizing your test cases.

A note on equivalence classes

Which is the better set of test cases?

'abba'    'abba'
'abcabc'   'cddc'
'abcdef'   'effe'
'aaaaa'    'fggf'
' '        'ghhg'
'a'    'jkkj'

(Neither set is comprehensive!)
Black box and glass box testing (functional and structural)

Functional testing (aka black-box testing):
View the software component as a “black box”.
Use its specification to formulate testing goals (design time).
Generation of black box test cases can be done by anyone who is familiar with the program specification.

Structural testing (aka glass-box testing):
Exercise all parts of the code, i.e., use the code to formulate testing goals.
Structural testing must be done by someone familiar with the code – a programmer.

Testing strategy: Start with functional testing and supplement with structural test cases as needed.

```python
# ADDITIONAL GLASS BOX TESTS?
'''(str) -> int'''
For example, conditionals, loops – boolean expressions

```testsNeeded(s):
# ADDITIONAL GLASS BOX TESTS?
'''(str) -> int'''
For example, conditionals, loops – boolean expressions
```def testsNeeded(s):
    # ADDITIONAL GLASS BOX TESTS?
    '''(str) -> int'''
    For example, conditionals, loops – boolean expressions
    if len(s) != 0:
        prev_char = s[0]
        dup_ct = 1
        high_ct = 1
    else:
        high_ct = 0
    for i in range(1, len(s)):
        if s[i] == prev_char:
            dup_ct += 1
        else:
            prev_char = s[i]
        if dup_ct > high_ct:
            high_ct = dup_ct
        dup_ct = 1
    return high_ct

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

TYPES OF PROGRAMMING ERRORS
• syntax - program language keywords, grammar
• runtime - TypeError, NameError, IndexError, etc.
• logical (semantic) – e.g., dynamic typing error, longest char string error, etc.

documentation/tests might have errors

✓ Write and test simple cases (docstring)
✓ Generate additional test cases incl. edge cases
✓ Generate more additional tests – use equivalence classes for coverage of various types of input and expected results
✓ Generate more additional tests as needed for “glass-box” testing

✓ Write and test simple cases
✓ Generate additional test cases incl. edge cases
✓ Generate tests from equivalence classes for expected input/results
✓ Generate additional tests as needed for “glass-box” testing

TEST: run the software component once for each test case. Compare actual outputs with predicted outputs. If there are discrepancies, locate the faults and fix them (debugging), and re-test.
Unit testing
-- look at one isolated component (e.g., function, but even a single line of code)

Integrative testing
-- looks at behavior of the whole (e.g., program, but function, system)

Regression testing
-- keep the tests and re-run them whenever the software is modified (e.g., debugging, revising, adding new functionality)

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:
→ systematic approach to formulating tests:
  • simple/basic examples
  • edge (boundary) conditions
  • for different types of expected input (equivalence classes)
  • for different types of expected results (equivalence classes)
  • glass 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.

TESTING
SUMMARY – WHY, WHAT, HOW
• Many aspects of a system can be tested (correct results, load testing, user experience, etc. – also algorithms, even project specification)
• CIS 210 focus: programs that run and produce correct results according to a project specification
• Thorough, systematic testing increases confidence in the software's reliability and exposes as many faults (bugs – syntax, runtime, logical) as possible.
• Testing can prove the presence of faults, but cannot prove the absence of faults.
• Testing that starts at design time supports optimal design, development, and deployment of reliable software.
• Thorough testing: simple, basic, edge cases, equivalence classes, glass box testing.
• Test the tests – e.g., simple test cases, multiple tests per equivalence class.
• Testing includes unit testing, then integrative testing, plus regression testing.
• Test early, test often – automate your testing to make it practical.
From Testing to Debugging – Finding and Fixing Bugs

(Novice) programming → better way

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

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

Try to repair bugs by haphazardly tinkering with code
→ repeat the failure; further testing
→ (sub-)unit testing helps contain the bug
→ keep a copy of last working version

Have difficulty breaking problems down into parts suitable for separate chunks of code
→ good program design/keep functions small

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 collections / mutable data types
✓ Testing and debugging

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, how).

Python is a programming language and Python is an interpreter (program)
Python Shell is a REPL (read-evaluate-print loop)
Python primitive elements: objects; values/attributes; type; memory location; memory management; garbage collection; immutable and mutable data types
Combining primitive elements: Expressions; expressions evaluate to a value; short circuit evaluation of boolean expressions; overloaded operators
Naming values: Variables/assignment - assignment statements are not expressions and do not return a value; namespaces – builtins and global [__main__]; scope; dynamic typing, strong typing, reference semantics.

Functions are an executable data type; what happens when a function – method – is called:
Activation record/stack frame added to call stack for local namespace; return address
Call-by-assignment parameter passing
Functions always return a value (sometimes None)
Functions sometimes have side effects
<table>
<thead>
<tr>
<th>CIS 210</th>
<th>Python toolkit so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming/Computer Science concepts, 2.</td>
<td>numeric data types (int, float) and operations (e.g., +, **, round, abs)</td>
</tr>
<tr>
<td>Turing-complete</td>
<td>string data type and operations (e.g., +, len, count, find)</td>
</tr>
<tr>
<td>What happens when an assignment statement is executed: memory allocation; reference semantics</td>
<td>Boolean data type and relational/Boolean operations (e.g., &lt;, and)</td>
</tr>
<tr>
<td>Binary representation of data</td>
<td>Python collections data types and operations – tuples, lists, dictionaries</td>
</tr>
<tr>
<td>Iterative algorithms; accumulator pattern; Monte Carlo algorithms, encoding/decoding, data analysis</td>
<td>data type coercion functions, e.g., str, int</td>
</tr>
<tr>
<td>Systematic testing (and debugging)</td>
<td>NoneType (None)</td>
</tr>
</tbody>
</table>

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.