CIS 210: What to Expect Looking Ahead

• **CIS 210 Midterm:** in class on Thursday February 6th
  --Will cover up to and including projects (homework) from Week 4 (due Monday Feb. 3rd)

Today’s Agenda:

• Testing
• Debugging
• Review (if time)

Thursday’s Agenda:
• Testing and Debugging (Continued)
• Midterm Review Problems
Midway Student Experience Survey:

Login to DuckWeb and select Course Surveys on the Main Menu page (it is a link, not a tab) and then select a link that says "Open the Course Surveys site". After being redirected into the CollegeNET system, your list of courses will appear. Students can click on Evaluate in the Action column on the right side of the page to fill out their survey.
>>> type(True)          a) <class 'bool'>
>>> type('True')
>>> type(true)          b) <class 'str'>
>>> type('true')
>>> type('false')      c) NameError: name is not defined
>>> type('False')
>>> type(False)
>>> type(false)
GOAL: HIGH QUALITY COMPUTER PROGRAMS –

REUSABLE/MAINTAINABLE

RELIABLE

(EFFICIENT - TIME, SPACE)

ARE WE MEETING THIS GOAL?
def testsNeeded(s):
    """(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:
            dup_ct += 1
        else:
            prev_char = s[i]
            if dup_ct > high_ct:
                high_ct = dup_ct
                dup_ct = 1
    return high_ct

>>> testsNeeded('abbcccd')
??
def testsNeeded(s):
    """(str) -> int

    Returns length of longest single-char string in s.
    """

    # program specification

    >>> testsNeeded('abcccddef')
    3
    >>> testsNeeded('')
    0
    >>> testsNeeded('abcdef')
    1
    ...
GOAL: HIGH QUALITY COMPUTER PROGRAM – REUSABLE/RELIABLE/EFFICIENT

ARE WE MEETING THIS GOAL?

→ style guidelines support development of reusable *and reliable* code
GOAL: HIGH QUALITY COMPUTER PROGRAM –

RELIABLE: program runs; produces correct output

*according to the specification*
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/UX, platforms
Goal: program runs; produces correct output according to problem specification

When programs are not reliable ...
Goal: **program that runs** and results in correct output according to problem specification

**TYPES OF PROGRAMMING ERRORS**

- **syntax** - program language keywords, grammar

  → program won’t run at all
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.

→ program starts to run and “crashes”
def mybad():
    ... ...
    greeting $ 'hello, world'
    short_greeting = greeting [0] + greeting[99]
    return short_greeting

>>> mybad()
??
def mybad():
    greeting = 'hello, world'
    short_greeting = greeting[0] + greeting[99]
    return short_greeting

>>> mybad()

Syntax Error – program won’t run
def mybad():
    
    greeting = 'hello, world'
    short_greeting = greeting[0] + greeting[99]
    return short_greeting

>>> mybad()
??
def mybad():
    '''
    greeting = 'hello, world'
    short_greeting = greeting [0] + greeting[99]
    return short_greeting
    >>> mybad()
    Traceback (most recent call last):
    File "<pyshell#49>", line 1, in <module>
    mybad()
    File "/Users/kfreeman/Documents/tryit.py", line 4, in mybad
    short_greeting = greeting [0] + greeting[99]
    IndexError: string index out of range
    >>>
    Runtime error
def testsNeeded(s):
    """(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:
            dup_ct += 1
        else:
            prev_char = s[i]
            if dup_ct > high_ct:
                high_ct = dup_ct
            dup_ct = 1
    return high_ct

# REUSABLE/RELIABLE??

>>> testsNeeded('abbcccd')
??

if dup_ct > high_ct:
    high_ct = dup_ct
    dup_ct = 1
return high_ct
def testsNeeded(s):
    # DOES THIS CODE DO WHAT
    # IT IS SUPPOSED TO DO?
    """(str) -> int
    # program specification

    Returns length of longest
    single-char string in s.

    >>> testsNeeded('abccCREMENT''
    3
    >>> testsNeeded('')
    0
    >>> testsNeeded('abcdef')
    1
    ...
def testsNeeded(s):
    '''(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:
            dup_ct += 1
        else:
            prev_char = s[i]
            if dup_ct > high_ct:
                high_ct = dup_ct
            dup_ct = 1
    return high_ct

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

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

When programs run ...

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

Programming – Types of Errors

- **syntax** – SyntaxError - incorrectly formatted code
  $\rightarrow$ program will not execute

- **runtime** – for example, NameError, IndexError, etc.
  $\rightarrow$ program will crash (or otherwise exit)

- **logical** – for example, incorrect implementation
  $\rightarrow$ program runs; incorrect result

**How can we detect logical errors?**
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)
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

we need ...
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:**
??

**Edge (boundary) conditions:**
??

**For different types of expected input:**
??

**For different types of expected output:**
??
def testsNeeded(s):
    # DOES THIS CODE DO WHAT
    # IT IS SUPPOSED TO DO?

    """(str) -> int
    Returns length of longest
    single-char string in s.
    """

    # program specification

    >>> testsNeeded('abcccdef')
    3
    # basic

    >>> testsNeeded('')
    0
    # edge

    >>> testsNeeded('abcdef')
    1
    # different types of input
    # different types of results

    ...
A systematic approach to formulating testing goals:

**Simple/Basic examples:**
For example, 'abccdef', 'ccbbcccccee'

**Edge (boundary) conditions:**
For example, empty string, string length 1

For different types of expected input:
For example, strings with no repeating chars, single char strings, long string at beginning, middle, end, ...

For different types of expected output:
For example, 0, 1, 2, 10
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?

(a) ‘abba’
    ‘abcabc’
    ‘abcdef’
    ‘aaaaa’
    ‘ ’
    ‘a’

(b) ‘abba’
    ‘cddc’
    ‘effe’
    ‘fggf’
    ‘ghhg’
    ‘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.
From last time...
✓ 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.
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
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)
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
            high_ct = dup_ct
    return high_ct
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 (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.
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
Midterm Prep:

The midterm from last term is posted on the course website.

I will post solutions early next week.
Midterm Prep:

The questions are not a comprehensive study guide! They are given here to provide a sense of the types of questions that may be on the CIS 210 midterm exam.

To prepare thoroughly for the midterm exam you should review projects and project solutions, quizzes, class notes, labs, and readings from the text.

The midterm exam will be given in-class, and will comprise multiple choice questions, short-answer questions, and questions where the solution will require you to write Python code according to the usual CIS 210 style guidelines.

No outside resources are allowed during the exam, with the exception of one 3x5” index card of handwritten notes.
def thrice(x):
    
    x += 1
    m = 3
    return m * x

>>> x = 5
>>> x
a) 5
>>> thrice(x)
??
d) 18
??
>>> m
b) 6
e) error
??
def thrice(x):
    """
    x += 1
    m = 3
    return m * x
    """
    a) 5
    b) 6

>>> x = 5
>>> x
5
>>> thrice(x)
18
c) 15
d) 18
e) error
What is printed when the following code is executed (after first time through the for loop)?

```
note = "We can't stop for gas, we're already late."
for char in note:
    print(char)
```

a) .  
b) e  
c) W  
d) We can't stop for gas, we're already late.”  
e) NameError: name 'char' is not defined
What is printed when the following code is executed (after first time through the for loop)?

```python
note = "We can't stop for gas, we're already late." 
for char in note:
    print(char)
```

a) .
b) e
c) W
d) We can't stop for gas, we're already late.”
e) NameError: name 'char' is not defined
What is the result when the following code is executed?

```python
>>> s = 'jello, world'
>>> s[0] = 'h'
```

a) s will have value ‘hello, world’

b) s will have value ‘j’

c) TypeError: 'str' object does not support item assignment

d) NameError: NameError: name 's' is not defined
What is the result when the following code is executed?

```python
>>> s = 'jello, world'
>>> s = 'h' + s[1:]
```

a) s will have value ‘hello, world’

b) s will have value ‘jello, world’

c) TypeError: 'str' object does not support item assignment

d) NameError: NameError: name 's' is not defined
What will be printed when the following Python code is executed?

```python
>>> isinstance(101, float) == True
??

>>> isinstance(101, float)
??

>>> False == False
??
```
What will be printed when the following Python code is executed?

```python
>>> isinstance(101, float) == True
False
```

```python
>>> isinstance(101, float)
False
```

```python
>>> False == False
True
```
Consider the following code:

```python
n = 5
mysum = 0
for ctr in range(1, n):
    mysum = mysum + ctr
print(mysum)
```

This code is an example of

a) accumulator pattern  b) TypeError  c) conditional
d) indefinite iteration  e) infinite loop
CIS 210: iClicker Question

What will be printed?

n = 5
mysum = 0
for ctr in range(1, n):
    mysum = mysum + ctr
print(mysum)
??

a) 15  b) 10  c) 24
What will be printed when the following Python code is executed?

```python
n = 5
mysum = 0
ctr = 1
while ctr:
    mysum = mysum + ctr
    ctr += 1
print(mysum)
```

This code is an example of
a) accumulator pattern   b) TypeError   c) conditional
d) indefinite iteration   e) infinite loop
Given the following Python code:

```python
0 import math
1 def isInCircle(x, y, r):
2     '''(number, number, number) -> ??
3     Returns True if point (x, y) is in
4     the circle with radius r.
5     >>> isInCircle(0, 0, 1)
6     True
7     >>> isInCircle(.5, .5, 1)
8     True
9     >>> isInCircle(1, 2, 1)
10    False
11     '''
12     d = math.sqrt(x**2 + y**2)
13     isIn = d <= r
14     return isIn
```

Complete the type contract: ??
Given the following Python code:

```python
import math

def isInCircle(x, y, r):
    """(number, number, number) -> ??
    Returns True if point (x, y) is in
    the circle with radius r.
    ""

    d = math.sqrt(x**2 + y**2)
    isn = d <= r

    return isn

>>> isInCircle(0, 0, 1)
True
>>> isInCircle(.5, .5, 1)
True
>>> isInCircle(1, 2, 1)
False
""
```

Complete the type contract:
"""(number, number, number) -> bool"""
Given the following Python code:

```
0 import math
1 def isInCircle(x, y, r):
2     """(number, number, number) -> ??
3     Returns True if point (x, y) is in
4     the circle with radius r.
5     >>> isInCircle(0, 0, 1)
6     True
7     >>> isInCircle(.5, .5, 1)
8     True
9     >>> isInCircle(1, 2, 1)
10    False
11 ""
12 d = math.sqrt(x**2 + y**2)
13 isIn = d <= r
14 return isIn
```

Which code would give the same results as `isInCircle` lines 12-14 (changes are in bold)?

a) \(d = \text{math.sqrt}(x**2 + y**2)\)
\[
\text{return } d = r
\]

b) \(d = \text{math.sqrt}(\text{pow}(x, 2) + \text{pow}(y, 2))\)
\[
\text{return } d \leq r
\]

c) \(d = \text{math.sqrt}(x**2 + y**2)\)
\[
\text{return } d < r
\]

d) \(d = \text{math.sqrt}(\text{pow}(x, 2) + \text{pow}(y, 2))\)
\[
\text{isIn} = d < r
\]
\[
\text{return } \text{isIn}
\]

e) \(d = \text{math.sqrt}(x**2 + y**2)\)
\[
\text{return } d
\]

Complete the type contract: ??
Given the following Python code:

def q29(s1):
    '''(str) -> str

    s2 = 
    for ch in s1:
        if ch not in s2:
            s2 += ch

    return s2

Which brief description is appropriate for q29?

a) copies s1 to s2; returns s2

b) copies all characters except the last character in s1 to s2; returns s2

c) copies 1st occurrence of each character in s1 to s2; returns s2

d) determines whether s1 is an empty string

e) creates and returns s2, a string of the characters that repeat (occur more than once) in s1
Given the following Python code:

```python
def fizzbuzz(n):
    '''(int) -> None

    Play fizzbuzz up to n.
    Results are printed during play;
    None value is returned
    '''
    for i in range(1, n+1):
        m3 = (i % 3) == 0
        m5 = (i % 5) == 0
        if m3 and m5:
            print('fizzbuzz')
        elif m3:
            print('fizz')
        elif m5:
            print('buzz')
        else:
            print(i)
    print('Game over!')
    return None
```

1. **for** is a Python **keyword**
2. **i** is a Python **identifier**
3. **m3** is a Python **primitive element**
4. **3** is a Python **primitive element**
5. **if** is a Python **keyword**
6. **'fizzbuzz'** is a Python **library module**
Given the following Python code:

```python
def fizzbuzz(n):
    '''(int) -> None

    Play fizzbuzz up to n.
    Results are printed during play;
    None value is returned"

    for i in range(1, n+1):
        m3 = (i % 3) == 0
        m5 = (i % 5) == 0

        if m3 and m5:
            print('fizzbuzz')
        elif m3:
            print('fizz')
        elif m5:
            print('buzz')
        else:
            print(i)

    print('Game over!')
    return None
```

- **1.** `for` is a Python keyword
- **2.** `i` is a Python identifier
- **3.** `m3` is a Python identifier
- **4.** `3` is a Python primitive element
- **5.** `if` is a Python keyword
- **6.** `'fizzbuzz'` is a Python primitive element (a string!)
- Testing and Debugging
- Midterm Q/A
- Automated Testing
- Techniques for Debugging
- A closer look at assignment – dynamic typing, strong typing
Automated testing:

- supports comprehensive testing
- communicates comprehensive testing
- supports regression testing
Automating 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

→ LET’S WRITE OUR OWN TEST FUNCTIONS
def ctemp_to_ftemp(ctemp):
    '''(number) -> float

    return fahrenheit temp (ftemp)
    equivalent to
given celsius temp (ctemp)

    >>> ctemp_to_ftemp(100)
    212.0
    >>> ctemp_to_ftemp(0)
    32.0
    >>> ctemp_to_ftemp(30)
    86.0
    >>> ctemp_to_ftemp(21.1)
    69.98
    '''
    ftemp = ctemp * 9/5 + 32
    return ftemp

def test_ctemp_to_ftemp():
    '''a very simple test function'''
    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_celsius_to_fahrenheit():
    
    return (ctemp_to_fahrenheit(100) == 212.0 and
            ctemp_to_fahrenheit(0) == 32.0 and
            ctemp_to_fahrenheit(30) == 86.0 and
            ctemp_to_fahrenheit(21.1) == 69.98)

def test_celsius_to_fahrenheit():
    
    if ctemp_to_fahrenheit(100) != 212.0:
        return False
    if ctemp_to_fahrenheit(0) != 32.0:
        return False
    if ctemp_to_fahrenheit(30) != 86.0:
        return False
    return True
def test_ctemp_to_ftemp():
    '''(number) -> float'''

    print('Checking 100 ... ', end='')
    result = ctemp_to_ftemp(100)
    if result == 212.0:
        print(' its value:', result, 'is correct!')
    else:
        print(' Error: has wrong value', result, 'expected 212.0.')
    return False

    if ctemp_to_ftemp(0) != 32.0:
        return False
    if ctemp_to_ftemp(30) != 86.0:
        return False

    return True
def test_celsius_to_fahrenheit():
    """(number) -> boolean ""

    print('Checking *100* ... ', end='')
    result = celsius_to_fahrenheit(*100*)
    if result == *212.0*:
        print(' its value', result, 'is correct!')
    else:
        print(' Error: has wrong value', result, 'expected *212.0* .')
    return False

...
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.