GOAL: HIGH QUALITY COMPUTER PROGRAMS –
REUSABLE/MAINTAINABLE
RELIABLE
(EFFICIENT - TIME, SPACE)

ARE WE MEETING THIS GOAL?

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

ARE WE MEETING THIS GOAL?

→ style guidelines support development of reusable and reliable code

```
def testsNeeded(s):
    # REUSABLE/RELIABLE?
    """(str) -> int
    Returns length of longest single-char string in s.
    # program specification
    >>> testsNeeded('abbcccd')
    3
    >>> testsNeeded('')
    0
    >>> testsNeeded('abcdef')
    1
    ""
    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
                else:
                    dup_ct = 1
        return high_ct
```
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 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

• runtime - TypeError, NameError, IndexError, etc.
  → program starts to run and “crashes”

```python
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()
    ???
    >>>
    Runtime error

>>> testsNeeded('abcccdef')
3
>>> testsNeeded('abcd')
1

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

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)

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
  basic examples of use
that reflect the project specification
automated testing (e.g., doctest.testmod)

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.

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 ...
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:

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" 'abba' 'abpbs' "ddc" 'abcd' 'oeef' 'aaaa' 'ffgg' 'a' 'gghg' 'b' 'jkkj'

(b) (Not 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
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

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

That's a lot of testing!
Test early, test often → automate testing to make it practical.
def thrice(x):
    x += 1
    m = 3
    return m * x

>>> x = 5
>>> x
5
>>> thrice(x)
15
>>> thrice(x)
18
>>> thrice(x)
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 the result when the following code is executed?

>>> 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

cis-210
What is the result when the following code is executed?

>>> s = 'jello, world'
>>> 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

• Testing  
• Debugging  
• Midterm Q/A  
• A closer look at assignment – dynamic typing, strong typing

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

→ LET’S WRITE OUR OWN TEST FUNCTIONS

```python
def ctemp_to_ftemp(ctemp):
    """(number) -> float
    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
```

```python
def test_ctemp_to_ftemp():
    """(number) -> 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
    return True
```

```python
def test_ctemp_to_ftemp():
    """(number) -> 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
    return True
```
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

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).

levels of abstraction

10 steps to debugging your code:

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

Always keep a Backup Copy of your code!
Given the following UNTESTED Python code:

```python
def isDivisible(n, m):
    return (n % m) == 0

def higherLevel(n, m):
    if isDivisible:
        print('yes')
    else:
        print('no')
    return None
```

```python
>>> isDivisible(15, 5)
>>> higherLevel(15, 5)
```

```python
>>> isDivisible(14, 5)
>>> higherLevel(14, 5)
```

```python
n = 15
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!')
```

```python
n = 15
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!')
```

✓ Testing
✓ Debugging
• Midterm Q/A
• A closer look at assignment – dynamic typing, strong typing
CIS 210
Assignment statements

Python is a dynamically typed language

```python
>>> a = 10
>>> type(a)
??
```

```python
>>> a = 'hello, world'
>>> type(a)
??
```

CIS 210
Assignment statements

Python is a dynamically typed language

```python
>>> a = 10
>>> type(a)
??
```

```python
>>> a = 'hello, world'
>>> type(a)
??
```

CIS 210
Assignment statements

Static, Dynamic – for example, scope, type

static – can be determined by reading code (only)

dynamic – scope/type is determined when code executes

CIS 210
Assignment statements

```python
static typing

var a : int
a = 4
a = 'hello'  X
```

CIS 210
Assignment statements

```python
dynamic typing

greenCt = 1
for ctr in range(4):
    greenCt = greenCt + 1
print(greenCt)
??
```

accumulator pattern

CIS 210
Assignment statements

```python
dynamic typing

greenCt = 1
for ctr in range(4):
    greenCt *= 1
print(greenCt)
??
```
A Structured Approach to Computational Problem-Solving

- review the project specification thoroughly
- write examples of expected results for specified inputs – re-review spec, if needed
- develop, review, and/or revise a problem-solving approach, using natural language, algorithm, pseudocode (not Python code)
- check algorithm using your examples – review algorithm, re-review spec, if needed

Starting with the lowest level function -
- write the function header
- write the function docstring – type contract
- write the function docstring – brief description
- write the function docstring – examples of use (use ones developed earlier)
- write the return statement

- using tools from the Python toolkit, start writing the body of the function
- test often, revise as needed
- test using examples in the docstring, and then project spec, and then others

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, indentation, comments, 1 line per command (or 1-2 for comments

Python is a programming language and Python is an interpreter (program)
Python Shell is a REPL (read-evaluate-print)

Python primitive elements: Objects - value/attributes, type
Combining primitive elements: Expressions - expressions evaluate to a value, short circuit evaluation of boolean expressions, controlled operation
Naming values: Variables/assignment - assignment statements are not expressions and do not return a value, namespaces - builtins and global (__main__); scope.

Functions are an executable data type; what happens when a function – method – is called:
Activation record/frame added to call stack for local namespace; return address called assignment parameter passing
Functions always return a value (return None)
Functions can have side effects

Functions as arguments
Iterative algorithms, accumulator pattern, Monte Carlo algorithms

Systematic approaches to testing and debugging, automated testing

Python toolkit so far
numeric data types (int, float) and operations (e.g., +, ** round, abs)
string data type and operations (e.g., +, len, count, find)
Boolean data type and operations (e.g., , and)
NoneType (None)
print expressions
Python Standard Library – math, turtle, random, doctest modules; import
assignment statement
Python repetition – for, while
Python conditionals (selection) – if
variable assignment
user-defined functions, function design, docstrings
IDLE interactive development environment, help function

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

CIS 210

CIS 210 Learning Outcomes