GOAL: HIGH QUALITY COMPUTER PROGRAM – RELIABLE, FAST (EFFICIENT) REUSABLE/MAINTAINABLE

ARE WE MEETING OUR GOALS?

→ CIS 210 style guidelines are an approach to computational problem solving that supports development of reliable, reusable code

→ design including examples of use -> testing
Goal: program that runs and results in correct output (according to problem specification)

Programming – Types of Errors

• syntax – for example, incorrectly formatted code (e.g. names) → SyntaxError; program will not execute

• runtime – for example, NameError, IndexError, etc. etc. → program will crash (or otherwise exit)

• logical – for example, misnamed vars, poor implementation → program runs; incorrect result

→ TESTING IS AN INTEGRAL PART OF PROGRAMMING

Testing

Integrative testing
→ looks at behavior of the whole program/system

Unit testing
→ look at one isolated component

Systematic approach to testing

• The aim of testing is to expose faults, so choose test cases that are likely to 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.

Testing to achieve high quality, reliable code

Many aspects to testing (Q/A, load test, user experience, ...)

Focus: programs that run and produce correct results according to a project specification

Unit testing; integrative testing

Errors – syntax, runtime, logical

Systematic approach – testing starts at design time

Consider test cases for various input values and expected results, including simple and “edge” values.
**Testing starts at program design time**

Function docstring:
- Type contract
- Short description of function including
description of parameter
  returned value
  side effects (if any)
- Examples of function execution/TESTS

```python
def ftemp_to_ctemp(ftemp):
    '''(number) -> float  TESTING AS PART OF
    PROGRAM DESIGN
    return celsius temp equivalent to
    input fahrenheit temp (ftemp)
    TEST CASES??
    '''
    ctemp = ftemp – 32 * 5/9
    ctemp = round(ctemp, 1)
    return ctemp
```

**Testing starts at design time**

1) Start with **simple cases** - can be done by hand.

2) Formulate a list of **testing goals**.

3) Generate one or more **test cases** (inputs, predicted outcomes) for each testing goal.

**Simple examples**: For example, 212, 32

Formulate a list of **testing goals**:

For different types of expected input:
- For example, temp > 212, temp < 32,
  32 <= temp <= 212

For different types of expected output:
- For example, low temp, high temp

Be sure to include boundary, or edge, conditions:
- For example, very high temp, very low temp
import doctest
def temp_to_ctemp(temp):
    """(number) -> float
    return celsius temp equivalent to input fahrenheit temp (temp)
    """
    ctemp = temp - 32 * 5/9
    ctemp = round(ctemp, 1)
    return ctemp

cis = temp_to_ctemp(212)
print(cis)  # 100

cis = temp_to_ctemp(32)
print(cis)  # 0

cis = temp_to_ctemp(75)
print(cis)  # 23.9

cis = temp_to_ctemp(300)
print(cis)  # 148.9

cis = temp_to_ctemp(24)
print(cis)  # -4.4

cis = temp_to_celsius(100)
print(cis)  # 37.8

cis = temp_to_celsius(0)
print(cis)  # 32.0

cis = temp_to_celsius(23.9)
print(cis)  # 7.7

cis = temp_to_celsius(148.9)
print(cis)  # 70.5

cis = temp_to_celsius(24)
print(cis)  # 13.3

Notice:

- 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.

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

- month in [2]
- month in [4, 6, 9, 11]
- month in [1, 3, 5, 7, 8, 10, 12]
- year is a leap-year
- year is a non-leap-year
- year is a turn-century-year
- days_in_month = 28
- days_in_month = 29
- days_in_month = 30
- days_in_month = 31

+ COMBINATIONS

Boundary values

Boundary 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.
- And: 0, 1, empty sequence, sequence length 1, ...

Write and test simple cases
Generate testing goals (equivalence classes)
Generate additional test cases incl. boundary

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.

Keep the test cases to run regression tests whenever the software is modified (e.g., debugging, revising, adding new functionality).

Black box and glass box testing

Functional testing (or 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 (or 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.

A Systematic Approach to Testing

Automating Testing
Test cases inside the function (e.g., docstring)

-- Advantages?

-- Disadvantages?

Writing separate test functions

TESTING SUMMARY – WHY, WHAT

- Thorough, systematic testing increases confidence in the software's reliability by exposing as many faults (bugs) as possible.
- Many aspects of a system can be tested (correct results, load testing, user experience, etc.)
- 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.
- Testing includes unit testing, integrative testing, regression testing.
- Equivalence classes plus glass box testing for thorough testing.
- Test the tests – e.g., simple test cases, multiple tests per equivalence class.
- Test early, test often – automate your testing to make it practical.

A Systematic Approach to Testing

TESTING SUMMARY – HOW

- Generate test cases against project specification
  - black box testing (functional/before code development):
    - consider input, output → generate simple test cases
    - consider input, output → generate equivalence classes
    - for each equivalence class → generate one to two test cases
    - consider input, output → generate edge test cases
  - glass box testing (structural/after code development):
    - consider all paths through the program → add ‘l tests
- Run the program for all test cases
- Regression testing – after any program modification, re-run all tests
For example,

Given a string, determine if it is a palindrome (reads the same backwards and forwards, e.g., ‘kayak’, ‘racecar’, ‘eye’)

→ equivalence classes
→ test cases

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
    return high_ct
CIS 210 Introduction to Computer Science Winter 2017

Programming/Computer Science concepts (Week 4)
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 primitive elements: Objects - value/attributes, operations, memory location (id).
Combining primitive elements: Expressions - expressions evaluate to a value.
Naming values: Variables/assignment - assignment statements are not expressions and do not return a value; namespace; variable scope.
User-defined functions – functions always return a value (sometimes None); functions sometimes have side effects.

What happens when a function is called?
- Activation record/stack frame added to call stack; local namespace
- Call-by-assignment parameter passing
- Side effects/returning a value

Iterative algorithms; accumulator pattern
Monte Carlo algorithms
Cipher encrypting and decrypting