GOAL: HIGH QUALITY COMPUTER PROGRAMS

REUSABLE/MAINTAINABLE
RELIABLE
(FAST/EFFICIENT - TIME, SPACE)

ARE WE MEETING THIS GOAL?

GOAL: HIGH QUALITY COMPUTER PROGRAM

REUSABLE/RELIABLE/EFFICIENT

ARE WE MEETING THIS GOAL?

def testsNeeded(s):
    # program specification
    Returns length of longest single-char string in s.

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

    # REUSABLE/RELIABLE?

    """(s: str) -> int"
    if len(s) == 0:
        return 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
                high_ct = max(high_ct, dup_ct)
            else:
                prev_char = s[i]
                if dup_ct > high_ct:
                    high_ct = dup_ct
                dup_ct = 1
        return high_ct

    # TESTED

    def testsNeeded(s):
        # REUSABLE/RELIABLE?
        """(s: str) -> int"
        if len(s) == 0:
            return 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
                    high_ct = max(high_ct, dup_ct)
                else:
                    prev_char = s[i]
                    if dup_ct > high_ct:
                        high_ct = dup_ct
                    dup_ct = 1
            return high_ct

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

    # REUSABLE/RELIABLE/EFFICIENT

    ARE WE MEETING THIS GOAL?

    ➤ style guidelines support development of reusable/maintainable and reliable code
GOAL: HIGH QUALITY COMPUTER PROGRAM

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

How do programs NOT meet this goal?

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()
```
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()
        OK?

    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>
        File "/Users/kfreeman/Documents/tryit.py", line 4, in mybad
        IndexError: string index out of range
        >>>
        Runtime error

    def testsNeeded(s):
        '''(s: str) -> int
        # REUSABLE/RELIABLE?
        Returns length of longest single-char string in s.
        # program specification
        >>> testsNeeded('abc\xc2\xa0cdef')
        3
        >>> testsNeeded('')
        0
        >>> testsNeeded('abcdef')
        1
        ...
GOAL: HIGH QUALITY COMPUTER PROGRAM
RELIABLE: program runs; produces correct output according to the specification

TYPES OF PROGRAMMING ERRORS
• syntax - program language keywords, grammar
• runtime - TypeError, NameError, IndexError, etc.
• logic/semantic - program runs and returns incorrect results

How can we detect logical errors?

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

→ style guidelines support development of reusable/maintainable and reliable 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)

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:

- 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?

(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.

```python
def testsNeeded(s):
    # ADDITIONAL GLASS BOX TESTS?
    # 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)

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

software engineering best practices:
→ systematic approach to formulating tests -- start early:
  • 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 -- test often:
  • 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

→ LET’S WRITE OUR OWN TEST FUNCTIONS

---

def ctemp_to_ftemp(cent):  # [ctemp: number] -> float
    return fahrenheit_temp(cent)

def test_ctemp_to_ftemp():  #() -> bool
    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():  #() -> bool
    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

---

def test_ctemp_to_ftemp():  #() -> bool
    if ctemp_to_ftemp(100) == 212.0:
        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
```python
def test_ctemp_to_ftemp():
    '''(number) -> bool'''
    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
```
10 steps to debugging your code:

Create a bug log (on paper, and/or by talking out loud, to LA or anyone):
1) record steps to reproduce the bug
2) describe what you expect to happen
3) describe what does happen
4) generate an idea about what is going wrong
5) test your idea — record what happens
6) repeat 4) and 5) as needed
7) when you have a good idea of what the bug is — BACKUP THE CODE
8) edit code to try to fix the bug
9) RECORD these changes to the code
10) test changes — the bug fix and regression testing (add new tests)

*Always keep a Backup Copy of your code!*

---

Given the following UNTESTED Python code:
```
def charCt(s, c):
    '''(s: str, c: str) -> int
    Return count of occurrences of char c in string s.
    >>> charCt("hello, world", 'o')
    2
    ct = 0
    for ch in s:
        if ch == c:
            ct += 1
    return ct
```

The set of test cases that will **NOT** find the bug in `charCt` is:

- a) `charCt('a', 'a')`
- b) `charCt('abc', 'a')`
- c) `charCt('abc', 'b')`
- d) `charCt('hi', 'o')`
- e) `charCt('hello, world', 'o')`
- f) `charCt('abc', 'c')`
- g) `charCt('ghi', 'x')`
- h) `charCt('x', 'x')`

---

Assignment statements

*Python is a dynamically typed language*

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

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

---

Assignment statements

*static typing*

```python
var a : int
a = 4
a = 'hello' X
```
Static, Dynamic – for example, scope, type

static – can be determined by reading code (only)
dynamic – scope/type is determined when code executes

Assignment statements

dynamic typing

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

accumulator pattern

Assignment statements

Recall:
dynamic typing (Python) v. static typing

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

Assignment statements

Recall:
dynamic typing (Python) v. static typing

var greenCt : int

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

Assignment statements

Recall:
dynamic typing (Python) v. static typing

var greenCt : int

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

Assignment statements

Recall:
dynamic typing (Python) v. static typing

var greenCt : int

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

def check(a: int, b: str) -> None:
    """(a: int, b: str) -> None
    Python 3.5 and higher
    ""
    print(a, b)
    return

>>> check("hello", 99)  # needs interpreter/IDE support
hello 99

def check(a: int, b: str, c: string) -> None:
    """
    print(a, b, c)
    return

Traceback (most recent call last): File "/Users/kfreeman/Documents/cis210W18/projects-W18/check.py", line 1, in <module>  def check(a\int, b\str, c\string): NameError: name 'string' is not defined
Recall: **expressions** are combinations of **values** (operands) and **operators**, that can be **evaluated** and return a **result**.

For example,
```python
>>> 99 + 10
109
>>> len('hello')
5
>>> str.center('** **', 10)
'** ** '
```

Python is a strongly typed language.
CS 210

Python toolkit (week 4)
- numeric data types [int, float] and operations (e.g., +, -, %, pow, round, abs)
- string data type and operations (e.g., +, len, count, find)
- boolean data type and operations (e.g., <, and)
- NoneType (None)
- expressions
- print, input
- variables (identifiers)
- assignment statement
- Python repetition – for (repeat), while
- Python conditionals (selection) - if
- user-defined functions; def, parameter list, docstring, return (values)

IDLE interactive development environment

Python introspection – help function

Python Standard Library – math, random, turtle, doctest modules; import

CS 210

A Structured Approach to Computational Problem-Solving
- review the project specification thoroughly
- 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, review as needed
- test using examples in the docstrings, and then project spec, and then others