CIS 210
• Recursion
• Testing

Define a function stars such that:

>>> stars(1)
*
>>> stars(2)
***
>>> stars(3)
*****
>>> stars(4)
***********

Do you see a pattern?

Each step of the problem can be described in terms of the prior step, i.e., a simpler version of the problem. → Recursive solution

What is (are) the most simple version(s) of the problem?
Or, What is (are) the base case(s)?

What should happen when it is not the base case?
Or, What is the recursive rule?

stars(1) *
stars(2) ***
stars(3) *****
stars(4) ***********

Do you see a pattern?

stars(1) *
stars(2) ***
stars(3) *****
stars(4) ***********

Each step of the problem can be described in terms of the prior step, i.e., a simpler version of the problem. → Recursive solution

What is the base case? → 1 star
What is the recursive rule? → n-1 pattern + n stars + n-1 pattern
def stars(n):
    '''
    //
    if n == 1:
        print('*')
    else:
        ??
    return None

What is recursion?

• defining things in simpler terms of themselves
• a problem solving approach where a task is divided into simpler and simpler versions of the original task (until it reaches a base case)
• a function that calls itself

Recursive function:

def countdown(n):
    ''' for loop is better for this problem '''
    for i in range(n, 0, -1):
        print(i)
    print('Blastoff!')
    return None

>>> countdown(4)
Recall: Factorials have a recursive definition

0! = 1  
1! = 1
2! = 2 * 1
3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1 = 4 * 3!

What is (are) the base case(s)?
Or, What is the most simple version of the problem?

What is the recursive rule?
Or, What should happen when it is not the base case?

---

Recall: Factorials have a recursive definition

0! = 1
1! = 1
2! = 2 * 1
3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1 = 4 * 3!

What is (are) the base case(s)?
What is the recursive rule?

---

def stars(n):
    '''
    # base case
    return('*')
    else:
        # recursive rule
        return stars(n-1) + ' ' + n * '*' + ' ' + stars(n-1)
Fractals: self-similarity at smaller and smaller scales
E.g., mathematical definitions, patterns in nature

-- Draw trunk
-- Recursively draw a tree on the right
-- Recursively draw a tree on the left
-- Stop when the tree is sufficiently small

CIS 210
def tree(n):
    """(int) -> None
    if n >= 0:
        forward(n)
        Draw a fractal tree with trunk and branches length n.
        Returns None.
    # go right
    right(30)
    >>> tree(100)
    # go left
    left(60)
    tree(n-15)
    >>> tree(100)
    # back to start
    back(n)
    tree(n)
    else: # base case
        return None
    return None

Why recursion?
• elegant approach to problem solving for problems with a recursive structure
  underlying algorithm is clear – solutions (programs) are simpler to write, analyze, and understand

Why not?
✓ can be prohibitively expensive

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):
    '''(str) -> int  # IT IS SUPPOSED TO DO?
    # DOES THIS CODE DO WHAT IT IS SUPPOSED TO DO?
    # program specification
    >>> testsNeeded('abccdef')
    3
    >>> testsNeeded('')
    0
    >>> testsNeeded('abcdef')
    1
    ...

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

When programs DON'T run ...

A survey of programming errors ("bugs")
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.

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

How can we detect logical errors?

→ documentation/tests might have errors
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)**

```
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('abccdef')
    3  # basic
    >>> testsNeeded('')
    0  # edge
    >>> testsNeeded('abcde')  # different types of results
    1
    ...  
```

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:** ??
- **Edge (boundary) conditions:** ??
- **For different types of expected input:** ??
- **For different types of expected output:** ??

```
A systematic approach to formulating testing goals:

Simple/Basic examples:
  For example, 'abccdef', 'aabbcdddee'

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?

'abba'  'abba'
'abcabc' 'cdcd'
'abcdef' 'effe'
'aaaaa' 'fggf'
' ' 'ghhg'
'aa' 'jkkl'

(Neither set is comprehensive!)

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.

Black box and glass box testing (functional and structural)

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.

# ADDITIONAL GLASS BOX TESTS?

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

def my_average(dataset):
    """(string) -> float

    Returns average of single digit numbers in dataset, but zeros are ignored. Return 0 if there is no real data.
    """
```python
count = 0
total = 0
for value in dataset:
    if value != '0':
        total += int(value)
        count += 1
avg = total / count
return avg
```