CIS 210

VARIABLE SCOPE

```python
def twice(x):
    '''
    y = 2
    result = y * x
    return result

>>> y = 5
>>> y
??

>>> twice(y)
??

>>> x
??
```

Python keeps track of variables using **namespaces** – a directory of names and objects.

**SCOPE** is where the namespace is accessible.
Python keeps track of variables using namespaces – a directory of names and objects.

SCOPE is where the namespace is accessible.

When we start Python, two namespaces are created – the built-in namespace and the global namespace.

When we create names (e.g., variables, function definitions) in a Python session, they are added to the global (main) namespace.

Each time Python executes a function, a local namespace is created inside the global namespace.
```python
CIS 210
SCOPE

def twice(x):
    # global namespace
    """
    y = 2; print(y)  # y is local to twice
    result = y * x   # when twice is executing
    return result
    """

>>> y = 5
>>> twice(y)  # local namespace – func exec
>>> y  # global again
```

```python
CIS 210
SCOPE

Python searches namespaces in this order:

Local, then

Global, then

Built-in
```

```python
CIS 210
SCOPE

def twice(x):
    # global namespace
    """
    x += 1  # x is local to twice
    m = 2
    return m * x
    """

>>> x = 5
>>> twice(x)  # shared variables outside of the function -> risky
>>> m  # global again
```
def test1(a):
    
    a += 5
    return a

>>> a = 6
>>> test1(a)
??

Namespaces:

- local (function execution)
- global (module)
- builtin (Python)

dir() shows a list of names in current scope
Define a function stars such that:

```python
>>> 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**

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

Do you see a pattern?

What is the most simple version of the problem? 
Or, What is the base case?

What should happen when it is not the base case? 
Or, What is the recursive rule?
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:
        stars(n-1)
        print(n * '*')
        stars(n-1)
    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:

```python
def countdown(n):
    '''
    if n == 0:
        # base case
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
    return None

>>> countdown(4)
```

```python
def stars(n):
    '''
    problem has recurring pattern → recursive solution'''
    if n == 1:
        print('*')
    else:
        stars(n-1)
        print(n * '*')
    return None
```

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

```python
def stars(n):
    '''
    problem has recurring pattern → recursive solution'''
    if n == 1:
        return('*')
    else:
        return stars(n-1) + (n * '*') + stars(n-1)
```
Recall: Factorials have a recursive definition

\[ 0! = 1 \]
\[ n! = n \times (n-1)! \]

What is the base case?
What is the recursive rule?

```python
def factR(n):
    if n == 0:
        # base case
        return 1
    else:
        # recursive call
        return n * factR(n-1)
```
def fact(n):
    '''not as elegant
    also less expensive'''
    result = 1
    for i in range(1, n+1):
        result *= i
    return result

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

```python
def testsNeeded(s):
    '''(str) -> int
    # DOES THIS CODE DO WHAT IT IS SUPPOSED TO DO?
    # IT IS SUPPOSED TO DO?
    # program specification
    # program runs
    # program does not have bugs (results are correct)
    # program handles the unexpected gracefully
    # program runs under extreme conditions
    # ... meets other specifications, e.g., HCI, platforms
    '''
    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
        else:
            prev_char = s[i]
        if dup_ct > high_ct:
            high_ct = dup_ct
    return high_ct

>>> testsNeeded('abccddf')
3
>>> testsNeeded('')
0
>>> testsNeeded('abcdef')
1
```
CIS 210
Goal: program runs; produces correct output according to problem specification

TYPES OF PROGRAMMING ERRORS
• syntax – for example, ??
• runtime – for example, ??
• logical – for example, ??
• documentation – for example, ??

How can we detect logical errors?

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) – e.g., dynamic typing error, reverse binary string error, longest char string error, etc.
→ documentation - e.g., of specification, tests

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**

---

**CIS 210**

**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.

---

**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
    """(str) -> int  # IT IS SUPPOSED TO DO?
    Returns length of longest
    single-char string in s.
    # program specification

>>> testsNeeded('abccccdef')
3  # basic
>>> testsNeeded('')  # edge
0  # different types of input
>>> testsNeeded('abcdef')  # different types of results
1

A systematic approach to formulating testing goals:

Simple/Basic examples:
For example, ‘abccccdef’, ‘abcdef’

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

For different types of expected input:
For example, strings with no dups, single char strings, strings with dups at beginning, middle, end, ...

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

A note on equivalence classes

Which is the better set of test cases?

'abba'    'abba'
'abcabc'   'cddc'
'abcdef'   'effe'
'aaaaa'    'fggf'
' '         'ghhg'
'a'        'jkkj'

(Neither set is comprehensive!)