Midterm (Thursday) Announcement:

1. You will be given a scantron and a seat assignment when you arrive.

2. You MUST bring ID to class.

3. You are allowed one 3 x 5 notecard with handwritten notes, no other outside sources.
Today’s Agenda

✓ Testing
✓ Debugging
• Tracing
• A closer look at assignment – dynamic typing, strong typing
import random

def roll(no_dice, dice_type):
    min = 1
    max = dice_type
    roll_again = "yes"

    while roll_again == "yes" or roll_again == "y":
        print("Rolling...")
        print("The values are....")
        for i in range(no_dice):
            print(random.randint(min, max))

        roll_again = input("Roll_again?")
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
From Testing to Debugging – Finding and Fixing Bugs

(Novice) programming → better way

Try to repair bugs by haphazardly tinkering with code
→ repeat the failure; further testing
→ (sub-)unit testing helps contain the bug
→ keep a copy of last working version

Have difficulty breaking problems down into parts suitable for separate chunks of code
→ good program design/keep functions small
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
Debugging – finding and fixing bugs

- **Look** at the code.
- Review/hand trace the code with a colleague, friend, pet, ...
- Try bits of code in the Shell.
- Isolate the bug (use print statements to find out where the program goes wrong).
- Split the code in half.
- **Change one thing at a time, for a reason.**
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!*
Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is easy to learn and has a tiny footprint with lightning fast performance. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows.
def isDivisible(n, m):
    """
    return (n % m) == 0
    """

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

Tests?
def isDivisible(n, m):
    """"
    return (n % m) == 0

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

>>> isDivisible(15, 5)
higherLevel(15, 5)  # incorrect invocation
??

>>> isDivisible(14, 5)
higherLevel(14, 5)  # incorrect invocation
??
n = 15
for i in range(1, n+1):
    m3 = (i % 3) == 0
    m5 = (i % 5) == 0
    if m3 and m5:
        print('fizzbuzz')
    if m3:
        print('fizz')
    if m5:
        print('buzz')
    else:
        print(i)
print('Game over!')
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!')
Given the following UNTESTED Python code:

```python
def charCt(s, c):
    '''(str, 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')`

b) `charCt('abc', 'a')`

c) `charCt('abc', 'a')`

d) `charCt('abc', 'b')`

e) `charCt('ghi', 'x')`

f) `charCt('hello, world', 'o')`

g) `charCt('x', 'x')`
Given the following UNTESTED Python code:

def charCt(s, c):
    """(str, str) -> int
    Return count of occurrences of char c in string s.
    """
    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')
   charCt('hi', 'o')
   charCt('hello, world', 'o')
b) charCt('abc', 'a')
   charCt('abc', 'b')
   charCt('ghi', 'x')
c) charCt('abc', 'a')
   charCt('def', 'b')
   charCt('x', 'x')
• closer look: Python objects/assignment
  ✓ dynamic typing
  – strong typing
  – Python object – value, type, id
  – reference semantics
  – garbage collection

lab
– formatted strings
– more namespaces:
if __ name__ ==

'__main__'
Assignment statements

Python is a dynamically typed language

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

Python is a dynamically typed language

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

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

static typing

```plaintext
var a : int
a = 4
a = 'hello' ✗ (not allowed)
```
Static, Dynamic – for example, scope, type

static – can be determined by reading code (only)

dynamic – scope/type is determined when code executes
• Dynamic Typing

Python is a **dynamically typed** language. This means that the Python interpreter does type checking only as code runs, and the type of a variable is allowed to change over its lifetime.

```python
>>> thing = "Hello"
>>> type(thing)
<class 'str'>

>>> thing = 28.1
>>> type(thing)
<class 'float'>
```
• Dynamic Typing

```python
>>> if False:
    1 + "two"  # This line never runs, so no TypeError is raised
else: 1 + 2

3

>>> 1 + "two"  # Now this is type checked
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```
• Static Typing

Static type checks are performed without running the program.
JAVA:

```java
public class HelloTypes {
    public static void main(String[] args) {
        String thing;
        thing = "Hello World";
        // thing = 42 would return an error!
        System.out.println(thing);
    }
}
```
Recall: we can combine objects in **expressions**, which are **evaluated** and **return a value**

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

**Python is a strongly typed language**
*(can’t do these operations on objects that are not strings)*
Recall: we can combine objects in expressions, which are evaluated and return a value

For example,

```python
>>> len(10)
??
>>> str.center(math.pi, 10)
??
```

Python is a strongly typed language
Recall: **expressions** are combinations of **values** (operands) and **operators**, that can be **evaluated** and **return a result**

For example,

```python
>>> 99.9 + 100
```

Python is an extremely strongly typed language
1) >>> x = 10
>>> x = 'hi'
>>> x = -99 < 0
2) n = 4
   x = 1
   for _ in range(20):
       approx_rt = .5 * (x + n/x)
   print(x)

3) >>> 'testing' + 123
   TypeError: must be str, not int
   a) dynamic typing
   b) operator overloading

4) >>> 1 + 123
   124
   >>> 'testing' + '123'
   'testing123'
   c) strong typing
   d) dynamic typing (error)
CIS 210

What are the following lines of codes examples of?

1) >>> x = 10
>>> x = 'hi'
>>> x = -99 < 0

ANS: a) approx_{rt} = .5 \times \left( x + \frac{n}{x} \right)

2) n = 4
x = 1
for _ in range(20):
    approx_{rt} = .5 \times (x + n / x)
    print(x)

ANS: d)

3) >>> 'testing' + 123
TypeError: must be str, not int

ANS: c)

4) >>> 1 + 123
124
>>> 'testing' + '123'
'testing123'

ANS: a) dynamic typing

b) operator overloading

c) strong typing

d) dynamic typing (error)
• closer look: Python objects/assignment
  ✓ dynamic typing
  ✓ strong typing
  – Python objects – value, type, id
  – reference semantics
  – garbage collection
Python overview – a closer look

Recall:
• Python has primitive elements

There is only one kind of primitive element in Python

OBJECTS
what are Python’s primitive elements?

<table>
<thead>
<tr>
<th>OBJECT</th>
<th>type</th>
<th>value(s)</th>
<th>id (memory location)</th>
</tr>
</thead>
</table>
PYTHON OBJECTS

- type: range of values and operations (attributes)
- value(s)
- id: memory location

Examples of Python objects:

```python
>>> 99
>>> 99.9
99
99.9

>>> type(99)
type(99.9)
<class 'int'>
<class 'float'>
```
Python overview – a closer look

✓ what are the primitive elements?
   -- object, with
     -- type that determines range of
       values and operations
       (attributes)
     -- value(s) of the object
     -- memory location
• closer look: Python objects/assignment
  ✓ dynamic typing
  ✓ strong typing
  ✓ Python objects – value(s), type, id
  – reference semantics
  – garbage collection
Recall: Objects can be named

For example,

```python
>>> x = 10
```
For example,

```python
>>> x = 10  
```

Python processes an assignment statement by:

1. evaluating the expression on the rhs
2. associating name on lhs with resulting value
Variable assignment

<variable> = <expression>

```python
>>> b = 20
>>> b
??
>>> b = 30
>>> b
??
>>> b = b + 1
>>> b ??
```
Variable assignment

\[
\text{<variable>} = \text{<expression>}
\]

```python
>>> b = 20
>>> a = b + 1
>>> a
```

??

1. evaluate the expression on the rhs
2. associate name on lhs with resulting value
1. evaluate the expression on the rhs
2. associate name on lhs with resulting value
1. evaluate the expression on the rhs
2. associate variable name on lhs with resulting value/Python object

- allocate space in memory for the object
- search current namespace – if name on lhs is not there:
  
  • assign name on lhs to address of memory location (*reference*)
  • add it to the current namespace

- if name on lhs is there:

  • replace old reference with new reference (address of memory location)
>>> b = 20
>>> id(b)
4297645024

 binnen (pointer)

>>> a = b + 1
>>> id(a)
4297645056

reference semantics:

the name on the lhs is a reference to the memory location of the data object
CIS 210

```python
>>> b = 20

semantics:
```

```python
>>> id(b)
4297645024
```

the name on the

```python
>>> a = b + 1
```

lhs is

a

```python
>>> id(a)
4297645056
```

to the memory

```python
>>> b
???
```

of the data object

```python
>>> a
???
```
CIS 210

>>> b = 20

**semantics:**

>>> id(b)
4297645024

the name on the lhs is

reference (pointer)

>>> a = b + 1

location

>>> id(a)
4297645056

>>> b
20

>>> a
21
1. evaluate the expression on the rhs
2. associate name on lhs with resulting value
>>> b = 20
    4297645024
>>> id(b)
          20
    4297645024

>>> a = b + 1

>>> b = 30
>>> b = 20
4297645024
>>> id(b)
20
4297645024

>>> a = b + 1
21
>>> id(a)
4297645056

>>> b = 30
>>> b = 20
>>> id(b)
4297645024

>>> a = b + 1
>>> id(a)
4297645056

>>> b = 30
>>> id(b)
4297645344
>>> b = 20
4297645024
>>> id(b)
4297645024
>>> a = b + 1
21
>>> id(a)
4297645056
>>> b = 30
30
>>> id(b)
4297645344
>>> b ??
>>> a ??
>>> b = 20
>>> a = b
>>> b = 40
>>> a

Evaluate rhs. **If it is a variable name:**

Let the variable on lhs refer to resulting object:

- **use address of rhs variable. no memory is allocated.**
- search current namespace – if name on lhs is not there:
  - assign name on lhs to address of memory location
  - add it to the current namespace

  – if name on lhs is there:
    - replace old reference with new reference (address)