

Lecture 11

Asserts and Debugging

Announcements for Today

(Optional) Videos

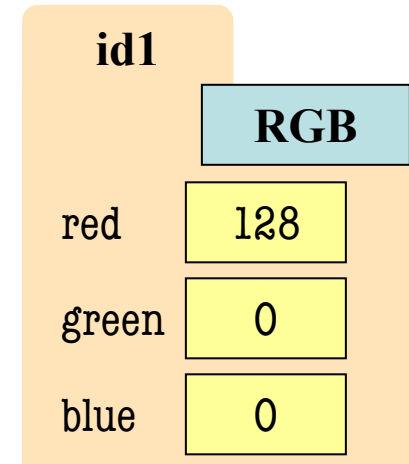
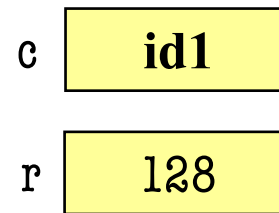
- **Lesson 13, 14** for **today**
- **Videos 15.1-15.7** next time
- **Prelim, Oct 15th 7:30-9:00**
 - Material up October 1st
 - Study guide is now posted
- **Conflict with Prelim time?**
 - Submit to Prelim 1 Conflict form sent by Corey
 - Do not submit if no conflict

Assignments

- A2 graded by Sunday
 - Still taking late submits
 - Also fill out survey
- A3 is now posted
 - Due Fri October 9th
 - Just before Fall Break
 - Graded when get back
 - Material is on exam

Using Color Objects in A3

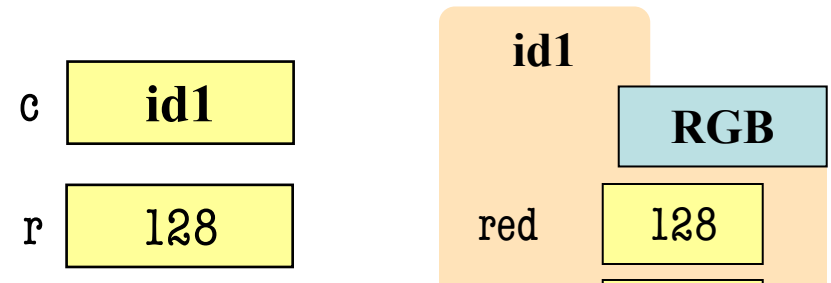
- New classes in introcs
 - RGB, CMYK, and HSL
- Each has its own attributes
 - **RGB**: red, blue, green
 - **CMYK**: cyan, magenta, yellow, black
 - **HSL**: hue, saturation, lightness
- Attributes have *invariants*
 - Limits the attribute values
 - Example: red is int in 0..255
 - Get an error if you violate



```
>>> import introcs
>>> c = introcs.RGB(128,0,0)
>>> r = c.red
>>> c.red = 500 # out of range
AssertionError: 500 outside [0,255]
```

Using Color Objects in A3

- New classes in introcs
 - RGB, CMYK, and HSL
- Each has its own attributes
 - **RGB**: red, blue, green
 - **CMYK**: cyan, magenta, yellow, black
 - **HSL**: hue, saturation, lightness
- Attributes have *invariants*
 - Limits the attribute values
 - Example: red is int in 0..255
 - Get an error if you violates



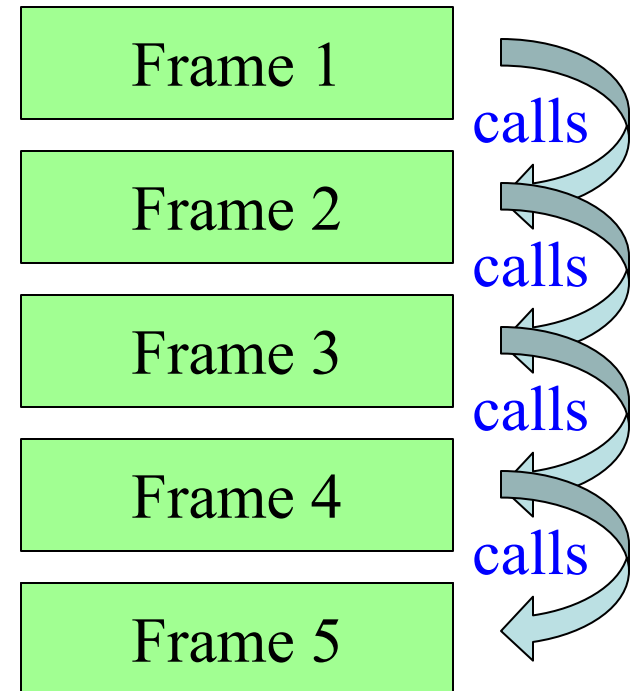
Constructor function.
To make a **new** color.

```
>>> import introcs
>>> c = introcs.RGB(128,0,0)
>>> r = c.red
>>> c.red = 500 # 0
AssertionError: 500
```

Accessing
Attribute

Recall: The Call Stack

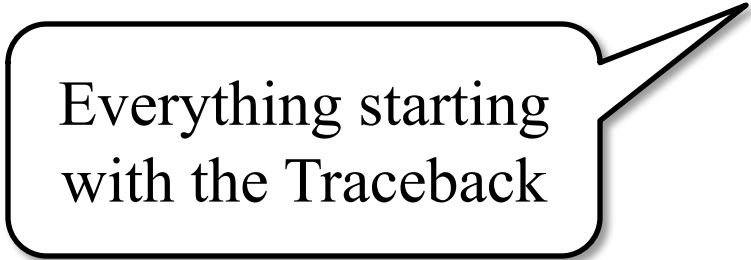
- Functions are **stacked**
 - Cannot remove one above w/o removing one below
 - Sometimes draw bottom up (better fits the metaphor)
- Stack represents memory as a ***high water mark***
 - Must have enough to keep the **entire stack in memory**
 - Error if cannot hold stack



Error Messages

Not An Error Message

ZeroDivisionError: division by zero



Everything starting
with the Traceback

An Error Message

Traceback (most recent call last):

File "error.py", line 41, in <module>

print(function_1(1,0))

File "error.py", line 16, in function_1

return function_2(x,y)

File "error.py", line 26, in function_2

return function_3(x,y)

File "error.py", line 36, in function_3

return x/y

ZeroDivisionError: division by zero

Errors and the Call Stack

```
# error.py
```

```
def function_1(x,y):
```

```
    return function_2(x,y)
```

calls

```
def function_2(x,y):
```

```
    return function_3(x,y)
```

calls

```
def function_3(x,y):
```

```
    return x/y # crash here
```

calls

```
if __name__ == '__main__':
```

```
    print(function_1(1,0))
```

Errors and the Call Stack

```
# error.py
```

```
def function_1(x,y):  
    return function_2(x,y)
```

```
def function_2(x,y):  
    return function_3(x,y)
```

```
def function_3(x,y):  
    return x/y # crash here
```

```
if __name__ == '__main__':  
    print(function_1(1,0))
```

Crashes produce the call stack:

Traceback (most recent call last):

File "error.py", line 20, in <module>
 print(function_1(1,0))

File "error.py", line 8, in function_1
 return function_2(x,y)

File "error.py", line 12, in function_2
 return function_3(x,y)

File "error.py", line 16, in function_3
 return x/y

Make sure you can see
line numbers in Pulsar.

Errors and the Call Stack

```
#
d
|   return function_2(x,y)
|
def function_2(x,y):
|   return function_3(x,y)
|
def function_3(x,y):
|   return x/y # crash here
|
if
```

Script code.
Global space

Where error occurred
(or where was found)

Crashes produce the call stack:

Traceback (most recent call last):

File "error.py", line 20, in <module>
 print(function_1(1,0))

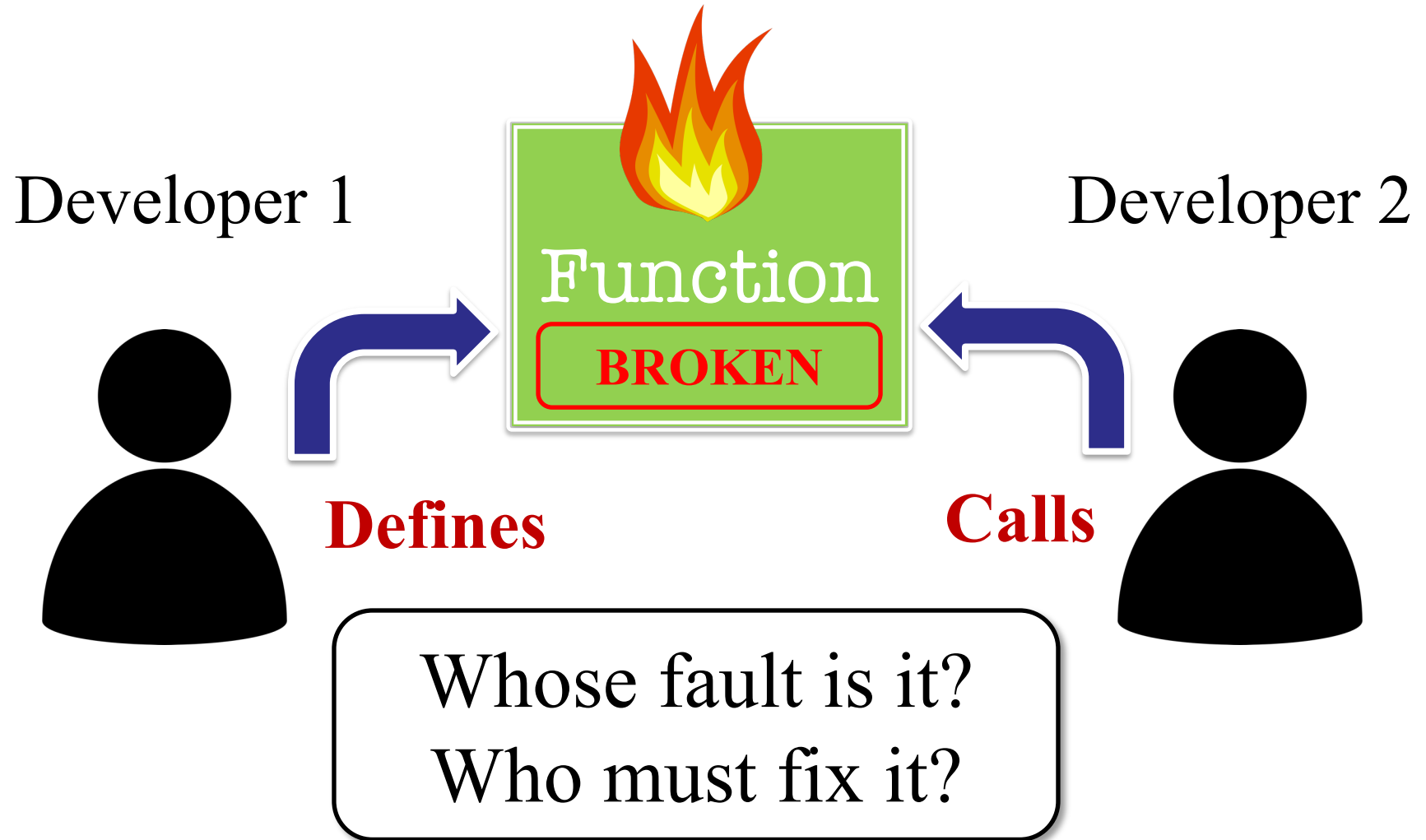
File "error.py", line 8, in function_1
 return function_2(x,y)

File "error.py", line 12, in function_2
 return function_3(x,y)

File "error.py", line 16, in function_3
 return x/y

Make sure you can see
line numbers in Pulsar.

Recall: Assigning Responsibility



Determining Responsibility

```
def function_1(x,y):  
    """Returns: result of function_2  
    Precondition: x, y numbers"""  
    return function_2(x,y)
```

```
def function_2(x,y):  
    """Returns: x divided by y  
    Precondition: x, y numbers"""  
    return x/y
```

```
print(function_1(1,0))
```

Traceback (most recent call last):

File "error1.py", line 32, in <module>
 print(function_1(1,0))

File "error1.py", line 18, in function_1
 return function_2(x,y)

File "error1.py", line 28, in function_2
 return x/y

ZeroDivisionError

Where is the error?

Approaching the Error Message

- Start from the top
- Look at function call
 - Examine arguments
 - (Print if you have to)
 - Verify preconditions
- Violation? Error found
 - Else go to next call
 - Continue until bottom

Traceback (most recent call last):

File "error1.py", line 32, in <module>

```
print(function_1(1,0))
```

File "error1.py", line 18, in function_1

```
return function_2(x,y)
```

File "error1.py", line 28, in function_2

```
return x/y
```

ZeroDivisionError: division by zero

Determining Responsibility

```
def function_1(x,y):  
    """Returns: result of function_2  
    Precondition: x, y numbers"""  
    return function_2(x,y)
```

```
def function_2(x,y):  
    """Returns: x divided by y  
    Precondition: x, y numbers"""  
    return x/y
```

```
print(function_1(1,0))
```

Traceback (most recent call last):

File "error1.py", line 32, in **A** module>
 print(function_1(1,0))

File "error1.py", line 18, in **B** function_1
 return function_2(x,y)

File "error1.py", line 28, in **C** function_2
 return x/y

ZeroDivisionError

Where is the error?

Determining Responsibility

```
def function_1(x,y):  
    """Returns: result of function_2  
    Precondition: x, y numbers"""  
    return function_2(x,y)
```

```
def function_2(x,y):  
    """Returns: x divided by y  
    Precondition: x, y numbers"""  
    return x/y
```

```
print(function_1(1,0))
```

Traceback (most recent call last):

File "error1.py", line 32, in <module>
 print(function_1(1,0))

File "error1.py", line 18, in function_1
 return function_2(x,y)

File "error1.py", line 28, in function_2
 return x/y

Error!

ZeroDivisionError: division by zero

Determining Responsibility

```
def function_1(x,y):
```

```
    """Returns: result of function_2
```

```
    Precondition: x, y numbers"""
```

```
    return function_2(x,y)
```

```
def function_2(x,y):
```

```
    """Returns: x divided by y
```

```
    Precondition: x, y numbs, y > 0"""
```

```
    return x/y
```

```
print(function_1(1,0))
```

Traceback (most recent call last):

File "error1.py", line 32, in <module>
 print(function_1(1,0))

File "error1.py", line 18, in function_1
 return function_2(x,y)

File "error1.py", line 28, in function_2
 return x/y

ZeroDivisionError: d

A

B

C

Where is the error?

Determining Responsibility

```
def function_1(x,y):  
    """Returns: result of function_2  
    Precondition: x, y numbers"""  
    return function_2(x,y)
```

```
def function_2(x,y):  
    """Returns: x divided by y  
    Precondition: x, y numbs, y > 0"""  
    return x/y
```

```
print(function_1(1,0))
```

Traceback (most recent call last):

File "error1.py", line 32, in <module>
 print(function_1(1,0))

File "error1.py", line 18, in function_1
 return function_2(x,y)

Error!

File "error1.py", line 28, in function_2
 return x/y

ZeroDivisionError: division by zero

Determining Responsibility

```
def function_1(x,y):  
    """Returns: result of function_2  
    Precondition: x, y numbs, y > 0"""  
    return function_2(x,y)
```

```
def function_2(x,y):  
    """Returns: x divided by y  
    Precondition: x, y numbs, y > 0"""  
    return x/y
```

```
print(function_1(1,0))
```

Traceback (most recent call last):

File "error1.py", line 32, in <module>

print(function_1(1,0))

Error!

File "error1.py", line 18, in function_1
 return function_2(x,y)

File "error1.py", line 28, in function_2
 return x/y

ZeroDivisionError: division by zero

This Makes Debugging Tricky

- Place error is detected != place error occurred
 - It can be further up in the call stack
 - It can even happen earlier in the code
 - Error messages are not enough to debug
- Debugging requires “**stepping-through**” code
 - You execute each line of code in your mind
 - You determine what Python does at that step
 - You verify that the two match (if not, bug)

Debugging Requires Visualization

Can Use Special Tools

Globals		Objects	
global		id1: function	
	last_name_first	last_name_first(n)	
			id1

Frames	
last_name_first	
n	"Walker White"
end_first	6
first	"Walker"
last	" White"
Return value	" White, Walker"

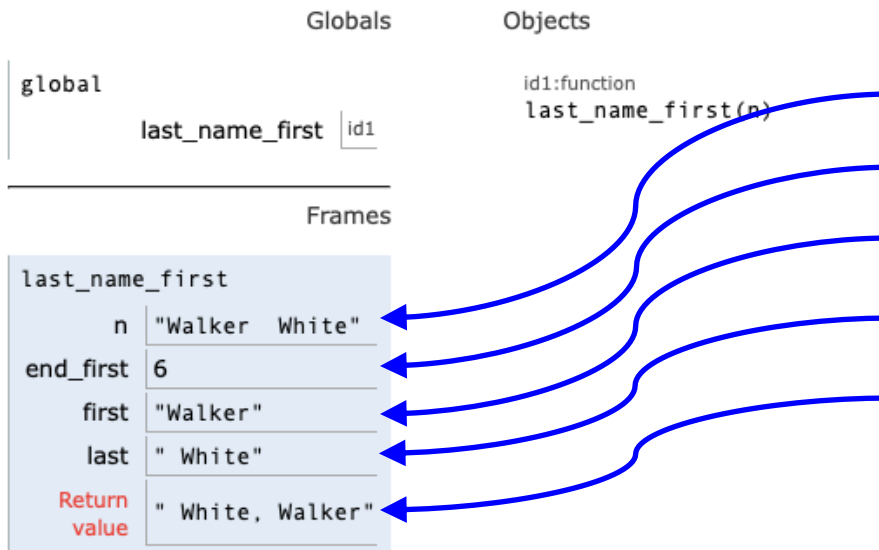
Can Use **print**

```
>>> last_name_first('Walker White')
Start last_name_first('Walker White')
end_first is "6"
first is "Walker"
last is " White"
result is " White, Walker"
Finish last_name_first('Walker White')

' White, Walker'
```

Debugging Requires Visualization

Can Use Special Tools



Can Use **print**

```
>>> last_name_first('Walker White')
Start last_name_first('Walker White')
end_first is "6"
first is "Walker"
last is " White"
result is " White, Walker"
Finish last_name_first('Walker White')

' White, Walker'
```

Case Study: Romanize

- Roman Numerals use letters instead of numbers
 - $I = 1$, $V = 5$, $X = 10$, $L = 50$, and $C = 100$
 - Can go higher, but will stop there for now
- Create other numbers by concatenation
 - $II = 2$, $III = 3$
 - $LV = 55$, $LVIII = 53$
- If a smaller number comes first, we subtract
 - $IV = 4$, $IX = 9$
 - $VL = 40$, $XCVL = 94$

Case Study: Romanize

```
def romanize(n):
```

```
    """
```

```
    Returns the Roman numeral for n
```

```
    Examples:
```

```
        romanize(14) returns 'XIV'
```

```
        romanize(75) returns 'LXXV'
```

```
        romanize(75) returns 'LXXV'
```

```
    Precond:  $0 < n < 100$  is an int
```

```
    """
```

```
    ...
```

Running the Tests: Bug #1

```
> > > romanize(14)
```

```
More than 10
```

```
numeralX: n = 1
```

```
choiceX = 'X'
```

```
tens = ''
```

```
romanize1to9: n = 4
```

```
Less than 5
```

```
romanize1to4: n = 4
```

```
choiceI = 'IV'
```

```
ones = 'IV'
```

```
'IV'
```

Running the Tests: Bug #1

```
> > > romanize(14)
```

```
More than 10
```

```
numeralX: n = 1
```

```
choiceX = 'X'
```

```
tens = ''
```

```
romanize1to9: n = 4
```

```
Less than 5
```

```
romanize1to4: n = 4
```

```
choiceI = 'IV'
```

```
ones = 'IV'
```

```
'IV'
```



Problem here

Running the Tests: Bug #1

```
> > > romanize(14)
```

```
More than 10
```

```
numeralX: n = 1
```

```
choiceX = 'X'
```

But also here

```
tens = "
```

```
romanize1to9: n = 4
```

```
Less than 5
```

```
romanize1to4: n = 4
```

```
choiceI = 'IV'
```

```
ones = 'IV'
```

Problem here

```
'IV'
```

Running the Tests: Bug #1

```
>>> romanize(14)
```

```
More than 10
```

```
numeralX: n = 1
```

```
choiceX = 'X'
```

```
tens = ''
```

```
romanize1to9: n = 4
```

```
Less than 5
```

```
romanize1to4: n = 4
```

```
choiceI = 'IV'
```

```
ones = 'IV'
```

```
'IV'
```

```
if n >= 50:
```

```
    print('More than 50')
```

```
    tens = numeralL(n//10)
```

```
elif n >= 10:
```

```
    print('More than 10')
```

```
    tens = numeralX(n//10)
```

```
    print('tens = '+repr(tens))
```

Running the Tests: Bug #1

```
> > > romanize(14)
```

```
More than 10
```

```
numeralX: n = 1
```

```
choiceX = 'X'
```

```
tens = ''
```

```
romanize1to9: n = 4
```

```
Less than 5
```

```
romanize1to4: n = 4
```

```
choiceI = 'IV'
```

```
ones = 'IV'
```

```
'IV'
```

```
if n >= 50:
```

```
    print('More than 50')
```

```
    tens = numeralL(n//10)
```

```
elif n >= 10:
```

```
    print('More than 10')
```

```
    tems = numeralX(n//10)
```

```
    print('tens = '+repr(tens))
```

Running the Tests: Bug #2

> > > romanize(75)

Expected: 'LXXV'

More than 50

numeralL: n = 7

Less than 90

numeralX: n = 2

choiceX = 'XX'

tens = 'LXX'

romanize1to9: n = 5

Between 5 and 8

romanize1to4: n = 0

choiceI = 'IV'

ones = 'VIV'

'LXXVIV'

Running the Tests: Bug #2

> > > romanize(75)

Expected: 'LXXV'

More than 50

numeralL: n = 7

Less than 90

numeralX: n = 2

choiceX = 'XX'

tens = 'LXX'

romanize1to9: n = 5

Between 5 and 8

romanize1to4: n = 0

choiceI = 'IV'

ones = 'VIV'

'LXXVIV'

Obvious
Problem

Running the Tests: Bug #2

```
> > > romanize(75)
```

```
Expected: 'LXXV'
```

```
More than 50
```

```
numeralL: n = 7
```

```
Less than 90
```

```
numeralX: n = 2
```

```
choiceX = 'XX'
```

```
tens = 'LXX'
```

```
romanize1to9: n = 5
```

```
Between 5 and 8
```

```
romanize1to4: n = 0
```

```
choiceI = 'IV'
```

```
ones = 'VIV'
```

```
'LXXVIV'
```

Obvious
Problem

But also why?

Running the Tests: Bug #2

```
> > > romanize(75)
```

```
Expected: 'LXXV'
```

```
More than 50
```

```
numeralL: n = 7
```

```
Less than 90
```

```
numeralX: n = 2
```

```
choiceX = 'XX'
```

```
tens = 'LXX'
```

```
romanize1to9: n = 5
```

```
Between 5 and 8
```

```
romanize1to4: n = 0
```

```
choiceI = 'IV'
```

```
ones = 'VIV'
```

```
'LXXVIV'
```

```
def romanize1to4(n):
```

```
    """
```

```
    Returns Roman numeral
```

```
    Ex: romanize(5) returns 'V'
```

```
    Precond: 0 < n < 5 is an int
```

```
    """
```

```
    print('romanize1to4: n = '+repr(n))
```

**Precondition
Violation!**

Running the Tests: Bug #2

```
> > > romanize(75)
```

```
Expected: 'LXXV'
```

```
More than 50
```

```
numeralL: n = 7
```

```
Less than 90
```

```
numeralX: n = 2
```

```
choiceX = 'XX'
```

```
tens = 'LXX'
```

```
romanize1to9: n = 5
```

```
Between 5 and 8
```

```
romanize1to4: n = 0
```

```
choiceI = 'IV'
```

```
ones = 'VIV'
```

```
'LXXVIV'
```

```
if n < 5:
```

```
    print('Less than 5')
```

```
    return romanize1to4(n)
```

```
elif n < 9:
```

```
    print('Between 5 and 8')
```

```
    return 'V'+romanize1to4(n-5)
```

```
else:
```

```
    print('Equal to 9')
```

```
    return 'IX'
```

Running the Tests: Bug #3

> > > romanize(60)

More than 50

numeralL: n = 6

Less than 90

numeralX: n = 1

choiceX = 'X'

tens = 'LX'

romanize1to9: n = 0

Less than 5

romanize1to4: n = 0

choiceI = 'IV'

ones = 'IV'

'LXIV'

Precondition
Violation

Running the Tests: Bug #3

> > > romanize(60)

More than 50

numeralL: n = 6

Less than 90

numeralX: n = 1

choiceX = 'X'

tens = 'LX'

romanize1to9: n = 0

Less than 5

romanize1to4: n = 0

choiceI = 'IV'

Also a
Violation

Precondition
Violation

ones = 'IV'

'LXIV'

def romanize1to9(n):

"""

Returns the Roman numeral for n

Ex: romanize(5) returns V

romanize(9) returns IX

Precond: 0 < n < 10 is an int

"""

print('romanize1to9: n = '+repr(n))

Running the Tests: Bug #3

```
> > > romanize(60)
```

```
More than 50
```

```
numeralL: n = 6
```

```
Less than 90
```

```
numeralX: n = 1
```

```
choiceX = 'X'
```

```
tens = 'LX'
```

```
romanize1to9: n = 0
```

```
Less than 5
```

```
romanize1to4: n = 0
```

```
choiceI = 'IV'
```

```
ones = 'IV'
```

```
'LXIV'
```

```
if n >= 50:
```

```
    print('More than 50')
```

```
    tens = numeralL(n//10)
```

```
elif n >= 10:
```

```
    print('More than 10')
```

```
    tens = numeralX(n//10)
```

```
    print('tens = '+repr(tens))
```

```
ones = romanize1to9(n % 10)
```

```
print('ones = '+repr(ones))
```

```
return tens+ones
```

Last Example Was a Nested Error

- Calling `romanize1to4(0)` is a bug
 - But it was caused by a call to `romanize1to9(0)`
 - Making changes to `romanize1to9` would be **bad**
- This type of error causes “**Doom Loops**”
 - You ask AI to fix a bug in your program
 - It finds an immediate bug, but not the source
 - Makes changes at the bug it found
 - This breaks specification, making code worse
 - You now have **more bugs**; repeat the process

Challenges With Our Approach

- This worked because the code was **simple**
 - We could read and understand it all
 - We could step through each line
- What do we do about **larger files**?
 - Assignment 7 will be > 500 lines of code
 - Cannot possibly step through it all
 - Too many print statements is hard to follow
- **Solution:** Dynamic Search
 - Adjust print statements until you find error

Dynamic Bug Search

- Start with a small number of print statement
 - Put them in the area you think error is
 - Leave large sections of code between prints
- Find the first print statement that is “wrong”
 - Error is must be before it (and after last print?)
 - Add more print statements in this area
 - Repeat this process as needed
- Problem is when cannot find anything wrong
 - Pick an area that was hard and add more prints

Aiding the Search Process

- In practice, searching is hard
 - Errors can go a long time until we catch them
 - Errors often seen by user, not developer
- Would be easier if errors just *crashed*
 - A crash always gives a stack trace
 - The user can just email that to you
- We do this with **assert statements**
 - Crash the program when an error is seen

Assert Statements

- **Form 1:** `assert <boolean>`
 - Does nothing if boolean is True
 - Creates an error if boolean is False
- **Form 2:** `assert <boolean>, <string>`
 - Very much like form 2
 - But error message includes the string
- Statement to **verify a fact is true**
 - Similar to `assert_equals` used in unit tests
 - But more versatile with complete **stack trace**

Why Do This?

- Enforce preconditions!
 - Put precondition as assert.
 - If violate precondition, the program crashes
- Provided code in A3 uses asserts heavily
 - First slide of lecture!

```
def exchange(from_c, to_c, amt)
    """Returns: amt from exchange
       Precondition: amt a float..."""
    assert type(amt) == float
    ...
```

Will do yourself in A4.

`assert <boolean>`

Creates error if <boolean> false

`assert <boolean>, <string>`

As above, but displays <String>

Example: Anglicizing an Integer

```
def anglicize(n):
```

```
    """Returns: the anglicization of int n.
```

```
    Precondition: n an int, 0 < n < 1,000,000"""
```

```
    assert type(n) == int, repr(n)+' is not an int'
```

```
    assert 0 < n and n < 1000000, repr(n)+' is out of range'
```

```
    # Implement method here...
```

Example: Anglicizing an Integer

```
def anglicize(n):
```

```
    """Returns: the anglicization of int n.
```

```
    Precondition: n an int, 0 < n < 1,000,000"""
```

```
    assert type(n) == int, repr(n)+' is not an int'
```

```
    assert 0 < n and n < 1000000, repr(n)+' is out of range'
```

```
    # Implement method here...
```

Check (part of)
the precondition

Error message
when violated

Aside: Using repr Instead of str

```
>>> msg = str(var)+' is invalid'
```

```
>>> print(msg)
```

```
2 is invalid
```

- Looking at this output, what is the type of var?

A: **int**

B: **float**

C: **str**

D: Impossible to tell

Aside: Using repr Instead of str

```
>>> msg = str(var)+' is invalid'
```

```
>>> print(msg)
```

```
2 is invalid
```

- Looking at this output, what is the type of var?

A: int

B: float

C: str

D: Impossible to tell

CORRECT

Aside: Using repr Instead of str

```
>>> msg = str(var)+' is invalid'
```

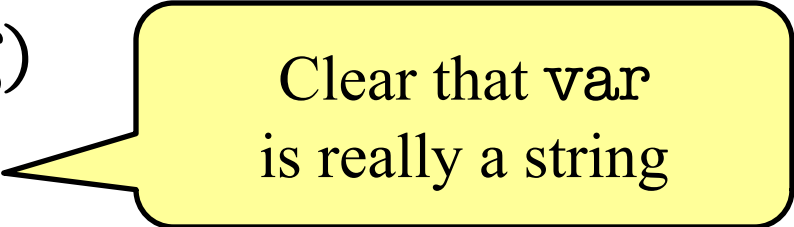
```
>>> print(msg)
```

```
2 is invalid
```

```
>>> msg = repr(var)+' is invalid'
```

```
>>> print(msg)
```

```
'2' is invalid
```



Clear that var
is really a string

Enforcing Preconditions is Tricky!

```
def lookup_netid(nid):
```

```
    """Returns: name of student with netid nid.
```

```
    Precondition: nid is a string, which consists of  
    2 or 3 letters and a number"""
```

```
    assert ?????
```

Assert use expressions only.
Cannot use if-statements.
Each one must fit on one line.

Sometimes we will
only enforce part of
the precondition

Enforcing Preconditions is Tricky!

```
def lookup_netid(nid):
```

```
    """Returns: name of student with netid nid.
```

```
    Precondition: nid is a string, which consists of  
    2 or 3 letters and a number"""
```

```
    assert type(nid) == str, repr(nid) + ' is not a string'
```

```
    assert nid.isalnum(), nid + ' is not just letters/digits'
```

Returns True if s contains
only letters, numbers.

Does this catch
all violations?

Using Functions to Enforce Preconditions

```
def exchange(src, dst, amt):
```

```
    """Returns: amount of src received.
```

```
    Precondition: src is a valid currency code
```

```
    Precondition: dst is a valid currency code
```

```
    Precondition: amt is a float"""
```

```
    assert ??????, repr(src) + ' not valid'
```

```
    assert ??????, repr(dst) + ' not valid'
```

```
    assert type(amt)==float, repr(amt)+' not a float'
```

Using Functions to Enforce Preconditions

```
def exchange(src, dst, amt):
```

```
    """Returns: amount of src received.
```

```
    Precondition: src is a valid currency code
```

```
    Precondition: dst is a valid currency code
```

```
    Precondition: amt is a float"""
```

```
    assert is_currency(src), repr(src) + ' not valid'
```

```
    assert is_currency(dst), repr(dst) + ' not valid'
```

```
    assert type(amt)==float, repr(amt)+' not a float'
```