

Using Color Objects in A3

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

c id1

r 128

id1	
	RGB
red	128
green	0
blue	0

```

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

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 Atom.

Errors and the Call Stack

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

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 Atom.

Assert Statements

```
assert <boolean> # Creates error if <boolean> false
assert <boolean>, <string> # As above, but displays <String>
```

- Way to force an error
 - Why would you do this?
- Enforce preconditions!
 - Put precondition as assert.
 - If violate precondition, the program crashes
- Provided code in A3 uses asserts heavily

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

Will do yourself in A4.

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

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(), repr(nid)+' is not letters/digits'
```

Returns True if s contains only letters, numbers.

Does this catch all violations?

Using Function to Enforce Preconditions

```
def exchange(curr_from, curr_to, amt_from):
    """Returns: amount of curr_to received.
    Precondition: curr_from is a valid currency code
    Precondition: curr_to is a valid currency code
    Precondition: amt_from is a float"""
    assert ?????, repr(curr_from) + ' not valid'
    assert ?????, repr(curr_to) + ' not valid'
    assert type(amt_from)==float, repr(amt_from)+' not a float'
```

Recovering from Errors

- try-except blocks allow us to recover from errors
 - Do the code that is in the try-block
 - Once an error occurs, jump to the catch

Example:

```
try:
    input = input() # get number from user
    x = float(input) # convert string to float
    print("The next number is "+str(x+1))
except:
    print('Hey! That is not a number!')
```

Annotations: "might have an error" points to the float conversion; "executes if error happens" points to the except block.

Try-Except is Very Versatile

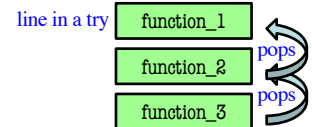
```
def isfloat(s):
    """Returns: True if string
    s represents a float"""
    try:
        x = float(s)
        return True
    except:
        return False
```

Annotations: "Conversion to a float might fail" points to the try block; "If attempt succeeds, string s is a float" points to the return True line; "Otherwise, it is not" points to the return False line.

Try-Except and the Call Stack

```
# recover.py
def function_1(x,y):
    try:
        return function_2(x,y)
    except:
        return float('inf')
def function_2(x,y):
    return function_3(x,y)
def function_3(x,y):
    return x/y # crash here
```

- Error "pops" frames off stack
 - Starts from the stack bottom
 - Continues until it sees that current line is in a try-block
 - Jumps to except, and then proceeds as if no error



Tracing Control Flow

```
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')

def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third.')

What is the output of first(2)?
```

Tracing Control Flow

```
def first(x):
    print('Starting first.')
    try:
        second(x)
    except:
        print('Caught at first')
    print('Ending first')

def second(x):
    print('Starting second.')
    try:
        third(x)
    except:
        print('Caught at second')
    print('Ending second')

def third(x):
    print('Starting third.')
    assert x < 1
    print('Ending third.')

What is the output of first(0)?
```