12. Logical Maneuvers

Topics:
- Loop-Body Returns
- Exceptions
- Assertions
- Type Checking
- Try-Except

Loop-Body Returns

Another way to terminate a loop.

Uses the fact that in a function, control is passed back to the calling program as soon as a return statement is encountered.

A Problem

Write a function

```python
MyFind(char, s)
```

that returns `True` if character `char` is in string `s` and returns `False` otherwise.

Typical While-Loop Solution

```python
def MyFind(char, s):
    k = 0
    while k < len(s) and char != s[k]:
        k = k + 1
    if k == len(s):
        return False
    else:
        return True
```

When the loop ends, if `k == len(s)` is True, then we never found an instance of char.

While-Loop Solution with a Loop-Body Return

```python
def MyFind(char, s):
    k = 0
    while k < len(s):
        if s[k] == char:
            return True
        k = k + 1
    return False
```

The function "jumps out of the loop" and returns `True` should it encounter an instance of char. If the loop runs to completion, that means there is no instance of char.
For Loop Solution with a Loop Body

```python
def MyFind(char,s):
    for k in range(len(s)):
        if s[k]==char:
            return True
    return False
```

The function “jumps out of the loop” and returns True should it encounter an instance of char. If the loop runs to completion, that means there is no instance of char.

Another For Loop Solution with a Loop Body

```python
def MyFind(char,s):
    for c in s:
        if c==char:
            return True
    return False
```

The function “jumps out of the loop” and returns True should it encounter an instance of char. If the loop runs to completion, that means there is no instance of char.

Boolean Variables

Review: Variables and floats

It is possible to assign a float value to a variable:

```python
a = 1.3
b = 10.1
c = 3.7
r = -b + math.sqrt(b*b-4*a*c))/(2*a)
```

Review: Variables and ints

It is possible to assign a string value to a variable:

```python
m = '7'
d = '4'
y = '1776'
date = m + '/' + d + '/' + y
```

Review: Variables and Booleans

It is possible to assign a boolean value to a variable:

```python
L = 1
R = 2
x = 1.3
inside = (L<=x) and (x<=R)
```
Boolean Variables

As the course progresses you will be dealing with logical situations that are increasingly complicated.

Boolean variables are a handy way of keeping track of what is going on.

Example: Leap Year

Gregorian Calendar Rule:

- \( Y \) is a leap year if it is a century year that is divisible by 400 or a non-century year that is divisible by 4.

Leap years: 1904, 2000, 2016

Not leap years: 1900, 2015

Example: Leap Year

Gregorian Calendar Rule:

- \( Y \) is a leap year if it is a century year that is divisible by 400 or a non-century year that is divisible by 4.

\[
\text{centuryYear} = (Y \% 100 == 0) \\
\text{if centuryYear:} \\
\quad \text{LY} = (Y \% 400 == 0) \\
\text{else:} \\
\quad \text{LY} = (Y \% 4 == 0)
\]

\( Y \) is a positive int.

LY is assigned the value True if \( Y \) is a leap year and False otherwise.

Boolean Functions

A function can return a boolean value.

This can be a handy way of encapsulating a complicated computation that culminates in the production of a True value or a False value.

Example: Intersecting Squares

Given two unit squares and a point, when is the point inside both squares?
Point in a Unit Square

Must have:
- \( a \leq x \leq a+1 \)
- \( b \leq y \leq b+1 \)

\((a,b),(a+1,b),(a,b+1),(a+1,b+1)\)

Point in a Unit Square

```python
def inS(a,b,x,y):
    """ Returns True if \((x,y)\) is inside the square with vertices \((a,b)\),\n    \((a+1,b),(a,b+1),\) and \((a+1,b+1)\). Otherwise, returns False."""
    xOK = (a <= x <= a+1)
    yOK = (b <= y <= b+1)
    z = (xOK and yOK)
    return z
```

Using inS

\[ z_2 = \text{inS}(a_1,b_1,x,y) \text{ and } \text{inS}(a_2,b_2,x,y) \]

\( z_2 \) is True if and only if \((x,y)\) is inside
(i) the unit square with lower left vertex \((a_1,b_1)\), and also
(ii) the unit square with lower left vertex \((a_2,b_2)\).

Exceptions

Exceptions are errors that occur while your program is running. The program stops running when an exception is "raised."

There are many types of exceptions.

Here are some examples...

**ValueError**

```python
>>> t = int('12F')
ValueError: invalid literal for int() with base 10: '123F'
```

In English:
The `int()` function does not accept a string unless it encodes a number.

**ImportError**

```python
>>> from superMath import sqrt
ImportError: No module named superMath
```

In English:
You cannot import stuff from a nonexistent module or a module that is not in the same working directory.
**ImportError**

```python
>>> from math import SquareRoot
ImportError: cannot import name SquareRoot
```

In English:
The math module does not contain a function named SquareRoot

---

**NameError**

```python
>>> x = 3
>>> x = y+2
NameError: name 'y' is not defined
```

In English:
The variable y does not exist.

---

**TypeError**

```python
>>> x = 3
>>> s = 'abc'
>>> t = s/x
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

In English:
You cannot divide a string by a number.

---

**TypeError**

```python
>>> from math import sqrt
>>> x = sqrt('a')
TypeError: a float is required
```

In English:
The square root function requires a number.

---

**ZeroDivisionError**

```python
>>> x = 3.0/0.0
ZeroDivisionError: float division by zero
```

In English:
Cannot divide by zero.

---

**Assertions**

They enable you to generate exceptions if something is wrong.

A good way to check that your code is doing what it should be doing.

A good way to focus on pre- and post-conditions during the program development phase.
Assertions: How They Work

Syntax:

```
assert B,S
```

B is a boolean expression.
S is a string.

If B is not true, then string S is printed and an exception is "raised".
Otherwise, nothing is done.

Checking Pre-, Post- Conditions

Typical:

1. At the start of a function body, are the preconditions satisfied?
2. At the end of the function body, does the value returned have the required properties?

Checking Pre-, Post Conditions

```python
def sqrt(x):
    """ Returns an approximate square root of x in that |L*L-x| <= .001
    PreC: x is a positive number. """
    assert x>0, 'The sqrt function requires a positive argument.'
    L = float(x)
    L = (L+x/L)/2
    L = (L+x/L)/2
    L = (L+x/L)/2
    assert abs(L*L-x)<=.001, 'Inaccurate Square Root'
    return L
```

Type Checking

Use `assert` and the function `isinstance`

How `isinstance` Works

It is a boolean-valued function with two arguments.

```
isinstance(x,int)
    True if variable x houses an int value
    Otherwise, False
isinstance(x,float)
    True if variable x houses a float value
    Otherwise, False
isinstance(x,str)
    True if variable x houses a string value
    Otherwise, False
```
Using `isinstance`

Guard against the user passing a string to `sqrt`:

```python
def sqrt(x):
    assert isinstance(x, float) or isinstance(x, int),
    print 'x must be type int or float'
```

The Try-except Construction

A graceful way to handle exceptions

Example: Try-Except

```python
try:
    from AintNoMath import sqrt
    print 'AintNoMath.sqrt unavailable'
except ImportError:
    from math import sqrt
    print 'AintNoMath.sqrt is not available'
# Code that uses sqrt...
a = 9; x = sqrt(a); print a, x
```

In the green code, if an `ImportError` exception is triggered, the mauve code is executed and `sqrt` comes from the `math` module. Otherwise, `sqrt` comes from `AintNoMath`.

Try-Except Construction

```python
try:
except NameOfException:
```

Code that may generate a particular exception

Code to execute if the particular exception is found

break

Another way to terminate a loop

But it must be used with care for style reasons.
How break Works

As soon as a break statement is executed inside a loop body, the loop ends and the next statement after the body is executed.

Example

Compute the smallest N so that N!>10

```python
fact = 1
for N in range(1,10000):
    fact = fact*N
    if fact>10:
        print N
        break
print fact
```

Loop range big enough to ensure we will get a large enough factorial

Recall that 5! = 1 x 2 x 3 x 4 x 5

Example

Print the smallest N so that N!>10

```python
fact = 1
for N in range(1,10000):
    fact = fact*N
    if fact>10:
        print N
        break
print fact
```

Bad Style! Have to guess a suitable for-loop range.

Example

Compute the smallest N so that N!>10

```python
fact = 1
N = 1
# fact = N!
while fact <=10:
    N = N+1
    fact = fact*N
print fact
```

While Loop Solution

A Good Example of break Usage

Consider the following problem.

A user enters an integer N from the keyboard and Python is to display the value of N!

Recall: 5! = 1x2x3x4x5 = 120

Use `math.factorial(N)`

A Good Example of break Usage

Possible issue.

When we use `math.factorial(N)`, the value of N must be nonnegative.

What if the user inputs -5?

Would like to say, “try again”
**A Good Example of break Usage**

```python
while True:
    N = raw_input('Enter pos int: ')
    N = int(N)
    if N>=0
        break
    else:
        print 'N must be nonnegative'
        print math.factorial(N)
```

Keep iterating until a nonnegative int is obtained.

---

**Another Issue**

If the user doesn’t enter a string of digits then the int statement will crash the program:

```python
N = raw_input('Enter pos int: ')
N = int(N)
```

This brings up the challenge of "exceptions" and "exception handling."

---

**A ValueError Exception**

```python
>>> int('12F')
ValueError: invalid literal for int() with base 10: '12F'
```

Exception a.k.a. run time error

---

**Challenge**

Is there a way we can keep soliciting keyboard input until the user enters a string of numbers?

Don’t want the program to terminate because of a ValueError.

---

**The Try-except Construction**

*A graceful way to handle exceptions*

---

**Example Showing Try-Except**

```python
from math import factorial

while True:
    n = raw_input('Enter an integer: ')
    try:
        n = int(n)
        break
    except ValueError:
        print 'Invalid input. Try again.'
    m = factorial(n)
    print m
```
from math import factorial

while True:
    n = raw_input('Enter an integer: ')
    try:
        n = int(n)
        break
    except ValueError:
        print 'Invalid input. Try again.'
print factorial(n)

If int(n) in the green block triggers a ValueError exception, then control passes to the cyan block. A message is printed and the loop continues.

If int(n) does not trigger a ValueError exception, then the break is executed and the loop is over and control passes to the print factorial(n) line.