12. Odds and Ends

Topics:
- floor, ceil, round, int
- a fact about string slicing
- more on in
- other ways of terminating a loop
- type
- try-except
- assert

math.floor, math.ceil, round, int

Let's look at what these functions do and the type of the value that they return.

<table>
<thead>
<tr>
<th>x</th>
<th>math.floor(x)</th>
<th>math.ceil(x)</th>
<th>round(x)</th>
<th>int(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.9</td>
<td>2.0</td>
<td>3.0</td>
<td>3.0</td>
<td>2</td>
</tr>
<tr>
<td>2.2</td>
<td>2.0</td>
<td>3.0</td>
<td>2.0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
<td>2</td>
</tr>
<tr>
<td>2.5</td>
<td>2.0</td>
<td>3.0</td>
<td>3.0</td>
<td>2</td>
</tr>
<tr>
<td>-3.9</td>
<td>-4.0</td>
<td>-3.0</td>
<td>-4.0</td>
<td>-3</td>
</tr>
<tr>
<td>-3.2</td>
<td>-4.0</td>
<td>-3.0</td>
<td>-3.0</td>
<td>-3</td>
</tr>
</tbody>
</table>

These functions all return values of type float:

- `math.floor(x)` largest integer <= x
- `math.ceil(x)` smallest integer >= x
- `round(x)` nearest integer to x

This function returns a value of type int:

- `int(x)` round towards 0

String Slicing
When String Slicing Goes “Beyond the End”

First, requesting a character from a position that doesn’t exist results in an error:

```python
s = 'abcdef'
t = s[10]
IndexError: string index out of range
```

On the other hand, requesting a slice that goes beyond the end of the “source string” is OK:

```python
0 1 2 3 4 5 6 7 8 9
s = 'abcdef'
t = s[4:10]
print t
```

A Handy Boolean Device

If `s1` and `s2` are strings, then

```python
s1 in s2
```

is a boolean-valued expression.

- **True** if there is an instance of `s1` in `s2`.
- **False** if there is NOT an instance of `s1` in `s2`.

in versus find

These are equivalent:

```python
x = s1 in s2
x = s2.find(s1)>=0
```

Type Checking With `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):
    if isinstance(x, str):
        print 'x must be type int or float'
        return
    L = x
    while abs(L - x/L) >= 10**-12:
        L = (L + x/L)/2
    return L
```

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
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.

Typical While-Loop Solution
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 return

```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 return

```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.

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 $N! > 10$

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

Recall that $5! = 1 \times 2 \times 3 \times 4 \times 5$

Example

Print the smallest $N$ so that $N! > 10$

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

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

While Loop Solution

Compute the smallest $N$ so that $N! > 10$

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

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: $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$

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

```
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
```

How It Works

```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
```

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.
How It Works

```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.'
print factorial(n)
```

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.

Note on Exceptions

The try-except block in the previous example was "looking for" ValueError exceptions

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

Python has a collection of exceptions and they all have names.

Examples of Exceptions

```
t = s[10]
IndexError: string index out of range
```
```
import simpleGraphics
ImportError: No module named simpleGraphics
```
```
x = y+1
NameError: name 'y' is not defined
```
```
s = s1/s2
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Try-Except Construction

```
try:
    Code that may generate a particular exception
except Name of Exception:
    Code to execute if the particular exception is found
```

Assertions

A graceful way to check that your program is doing what it should be doing

```
1. At the start of a function body, are the preconditions satisfied?
2. At the end of the function body, is the value returned the right type?
```

Assert

A handy debugging tool.

Used to check that things are "ok" at a particular point during execution.
**Assertions: How They Work**

assert B, S

If boolean expression B is not true, then string S is printed and an exception is generated.

**Checking Precondition**

```python
def sqrt(x):
    assert x>0, 'must have x>0'
    L=float(x);
    W=1.0
    while abs(L-W)/L > 10**-12:
        L = (L+W)/2
        W = x/L
    return L
```