Defensive Programming

CMPT 145

Defensive Programming

Prevent bugs through

- Avoiding bad use of the programming language
- Using best practices for programming

Defensive Programming

- Make sure that a program protects itself against incorrect or illegal data
- Assume Murphy's Law is true: Whatever can go wrong, will go wrong.
    - Add code to check those things that can't happen.
- Check the value(s) returned by functions

Pre- and Post-Conditions

- We've already used these in our interfaces.
- At the start of every function, check that the pre-conditions are true.
- This is part of the function, not testing.
- Unit testing should check post-conditions.

Checking pre-conditions

- Python has a tool for this: `assert`

```
def factorial(n):
    assert n >= 0, 'invalid input to factorial'
    if n <= 1:
         return 1
     else:
         return n*factorial(n-1)
```

- Syntax: `assert` *condition, optional-message*
- Causes Python to halt program execution with a run-time error
- Assertions can be turned off
- Use these for wolf-fencing too!

Checking pre-conditions

- Python has more general tool for this: `raise`

```
def fib(n):
    assert n >= 0, ’invalid input to fib’
    if n == 0 or n == 1:
        return n
    elif n > 20:
        raise Exception(’n = ’+str(n)+’ too big’)
    else:
        return fib(n-1) + fib(n-2)
```

- Syntax: `raise Exception(message)`
- Causes Python to halt program execution with a run-time error
- Exceptions cannot be turned off
- More about Exceptions later.

Don't shadow Python functions

- Shadow: When you use a variable name that's defined elsewhere in scope.
- Example: `sum` is a Python function, but we can still do:

```
sum = 0
for val in my_list:
    sum += val
```

- Python names for functions are not special.
- A name in Python has a value; sometimes the value is a function.

Know run-time exceptions

- You're not a beginner anymore.
- When Python halts your program, it tells you why.
- Know the names and potential causes for all run-time errors:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: list assignment index out of range
```

- This is not a burden. It is valuable information.
- Google for Python exceptions. Visit Stack-Overflow.

Know scoping

- What variables are in scope?
- What variables are shadowing others?
- Is this local or global?

```
x = 100

def trivial_function():
    print(x)
    x = x + 1

trivial_function()
```

Know scoping

- What variables are in scope?
- What variables are shadowing others?
- Is this local or global?

```python
x = 100

def trivial_function(x):
    print(x)
    x = x + 1

trivial_function(x)
```

List iteration

- Don't change the structure of your list while iterating
- Changing the contents is fine:

```
a = list(range(10))
i = 0
while i < len(a):
    a[i] = a[i] + 1
    i += 1
```

- Changing the structure is not good:

```
a = list(range(10))
i = 0
while i < len(a):
    del a[i]
    i += 1
```

Watch out for equality of floating point

- Floating point calculations have tiny errors due to finite precision.
- Every calculation adds a little more error
- Some calculations add a lot more error
- Two values are hardly ever equal.
- Poor:

```
x = 0.3
y = 0.1 + 0.1 + 0.1
if x == y:
    print(’Equal’)
else:
    print(’Not equal!’)
```

Watch out for equality of floating point

- Floating point calculations have tiny errors due to finite precision.
- Every calculation adds a little more error
- Some calculations add a lot more error
- Two values are hardly ever equal.
- Not a bad for large values:

```
x = 0.3
y = 0.1 + 0.1 + 0.1
if abs(x - y) < 0.000001:
    print(’Equal enough’)
else:
    print(’Not equal enough!’)
```

Watch out for equality of floating point

- Floating point calculations have tiny errors due to finite precision.
- Every calculation adds a little more error
- Some calculations add a lot more error
- Two values are hardly ever equal.
- Better for very small numbers, relative to 0.000001:

```python
x = 0.3
y = 0.1 + 0.1 + 0.1
if abs(x - y)/max(abs(x),abs(y)) < 0.000001:
    print(’Equal enough’)
else:
    print(’Not equal enough!’)
```

Watch out for division

- Difference between integer and floating point division
- Division by zero

```
x = 0.3
y = 0.1 + 0.1 + 0.1
if abs(x - y)/max(abs(x),abs(y)) < 0.000001:
    print(’Equal enough’)
else:
    print(’Not equal enough!’)
```

Watch out for division by zero

- Use an if-statement if you are setting the denominator yourself.

```
x = 0.3
y = 0.1 + 0.1 + 0.1

if x == 0 and abs(y) < 0.000001:
    print(’Equal enough’)
elif y == 0 and abs(x) < 0.000001:
    print(’Equal enough’)
elif abs(x - y)/max(abs(x),abs(y)) < 0.000001:
    print(’Equal enough’)
else:
    print(’Not equal enough!’)
```

Watch out for division by zero

- Use an assertion if the denominator value comes from some other part of the program.

```
def ratio(x, y):
    """Compute x/y for some reason.
    Preconditions: y != 0
    """
    assert abs(y) > 0, ’denominator zero in ratio’
    return x/y
```

Watch out for division by zero

- If you know why `ratio` is needed, you could avoid `assert`.

```python
def ratio(x, y):
    """Compute x/y for some reason.
    If y == 0, ratio returns 0 for some reason.
    Preconditions: x, y are numbers
    """
    if y == 0: return 0
    else: return x/y
```

- This has to be appropriate for your application!