Lecture #8: Efficiency vs Readability

Recap: Mutable Data Types
- Certain data types in python are mutable:
  - List, set
- Other data types in Python are immutable
  - Tuples
  - Primitive data types: integer, long, float, string, bool
- Dictionary:
  - Dictionary keys must be immutable
  - Dictionary values can be mutable or immutable

List Mutability

```
x = [1,2,3,4,5]
x[1] = 10
```

What will the following code do?
```
x = (1, 2, 3)
x[0] = 10  # What will this do?
d = {}  
key = [1, 2]  
value = [3, 4]  
d[key] = value  # What will this do?
```

Recap: Mutable Data Types

Mutability: Quick Diagram

```
B is assigned to A  
(B = A)

A is immutable  
(fst, str, tuple)
A is mutable  
(lst, dict, user-defined type)

A doesn’t change if B changes
B is modified in-place

A doesn’t change  
A also changes
```

Mutability: How it Works

Reference-type Storage  
``` 
Memory
Reference var
object on the managed heap
```

Value-type Storage  
``` 
Memory
object stored "in-line"
```

Notice that the reference-type "refers" to an object somewhere else in memory, namely, the managed heap. On the other hand, value-type objects (in most cases) are stored directly in the current, working memory.
Mutability: Why?

- Programming is a compromise between understandability and efficiency
  - Humans want to read and understand and maintain
  - Computers works the way they work

- Example:
  Passing a string to a function by reference or by copying.

Which one is more efficient for large strings?
Which one is probably more intuitive?

Recap: Mutability

Recap: Exceptions

- Python raises an exception whenever an error occurs:
  - ZeroDivisionError
  - IndexError

- Python handles errors by terminating immediately and printing an error message.
- Exceptions can be handled by the program, preventing a crash (next slide)
- Programs can also raise exceptions of their own (later in the course)

Recap: Handling Exceptions

- Using try statement with except clause to prevent program crash.
- The following program won’t crash even if you divide by 0:

```python
def safe_divide(x, y):
    quotient = "Error"
    try:
        quotient = x/y
    except ZeroDivisionError:
        print("Can't divide by zero!")
    return quotient

Result = safe_divide(3,0)
print("Result is: ", Result)
```

Can’t divide by zero!
Result is: Error

Recap: Variable Scope

```
a_var = 'global value'
def a_func():
    global a_var
    a_var = 'local value'
    print(a_var, 'a_var inside a_func()')
    print(a_var, 'a_var outside a_func()')
a_func()
```

Output?
global value [ a_var outside a_func()
local value [ a_var inside a_func()
local value [ a_var outside a_func() ]

Recap: Variable Scope

Why Exceptions?

- Exceptions are raised by the CPU and the operating system or by the program.
- Examples:
  - Division by Zero
  - File not Found
- More exceptions types:
  https://tinyurl.com/n2yhry

- Exceptions allow to pass the condition on to the calling function for proper handling.
More on Variable Scope

```python
a_var = 'global variable'
def len(in_var):
    print('called my len() function')
    l = 0
    for i in in_var:
        l += 1
    return l
def a_func(in_var):
    len_in_var = len(in_var)
    print('Input variable is of length', len_in_var)
a_func('Hello, World!')
Output?
```

Sequences

- A sequence has:
  - a finite length,
  - is empty when it has length 0,
  - is indexed by a positive integer, with the first element being 0.

- Examples:
  - Lists
  - Tuples
  - Strings

- Not: dictionary (no indexing)

Iterables

- Any object that you can use a for loop over
- Sequence => Iterable (not both ways)
- Examples:
  - Lists
  - Strings
  - Tuples
  - Dictionaries

- Functions that return special data types
  - Range
  - Zip
  - Map

Are these data types sequences or iterables?

Sequence vs Iterable

```python
>>> x = range(10)
>>> y
range(0, 10)
>>> len(x) # We can get the length
10
>>> x[5] # We can index
5
```

```python
>>> y = map(lambda x: x**2, [1, 2, 3])
>>> y
<map object at 0x101a3cb38>
>>> len(y) # We can’t get length Error!
>>> y[5] # We can’t index
```

Iterables: Why?

- Lazy evaluation: Each value is computed on demand. No all values have to be stored in memory!
- If we want to save a value, we need to either bind it to a variable or loop

Allows us to work with huge amounts of data!
Generators: Why?

- Generators return iterables and can be of infinite length.

```python
def naturals():
    i = 1
    while True:
        yield i
        i += 1

>>> for elem in naturals():
...     print(elem)
...
1
2
3
(keeps going, never ends)
```

Conclusion

Mutability, Scoping, Exceptions, Sequences, Iterables, and Generators:
- The computer does not need them
- Decades of practice in programming have shown: Humans need them. The resulting code is better.

More on these: In the labs.

- Next lectures: Object Oriented Programming
  (they say a biologist invented it)