Computational Structures in Data Science

Lecture #6: Mutability, Nonlocal, Exceptions

March 2nd, 2018

http://inst.eecs.berkeley.edu/~cs88

Computation Concepts today
• Mutability and Mutable Data Types
• Mutability and Nonlocal
• Exception and Exception Handling

What is Mutation?
• Mutation is the changing of value
• A mutable data type can be changed after it is created.

What will the following code do?

```python
x = [1,2,3,4,5]
x[1] = 10
x[4] = 50
x = [60,70]
```

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

Mutable Data Types:

List Mutability
```python
x = [1,2,3,4,5]
x[1] = 10
x[4] = 50
x = [60,70]
```

What will the following code do?
```python
d = {}
key = [2, 3]
value = [3, 4]
d[key] = value # What will this do?
```

Mutability is Tricky
• Mutability can often lead to unexpected behavior when writing program
• Example:
  ```python
  x = [1, 2, 3, 4]
y = x
  print(x[0])
  print(y[0])
  x[0] = 10
  print(x[0])
  print(y[0])
  ```
  • Both variables refer to the same list in the above example
  • It’s easy to mistake x and y as being two different lists
Mutability Example: List Creation

• Which variables point to the same list:
  \[ x = [1, 2, 3, 4] \]
  \[ y1 = x \]
  \[ y2 = \text{list}(x) \]
  \[ y3 = x[:] \]
  \[ y4 = \{\text{elem for elem in x}\} \]

Mutability Example: List Creation

• Which variables point to the same list:
  \[ x = [1, 2, 3, 4] \]
  \[ y1 = x \]
  \[ y2 = \text{list}(x) \]
  \[ y3 = x[:] \]
  \[ y4 = \{\text{elem for elem in x}\} \]

• \text{list} constructor function creates a copy of a list
• List comprehension always creates a new list.
• \[ x[:] \] also creates a copy of \[ x \]

Mutability Example: Appending to a list

• Which variables point to the same list?
  \[ x = [1, 2, 3, 4] \]
  \[ x.\text{append}(5) \]
  \[ y = x \]
  \[ y == [6] \]
  \[ z = x \]
  \[ z = z + [7] \]

Mutability Example: Nested lists

• Nested List: list of lists
• Example:
  \[ x = [1, 2, 3, 4] \]
  \[ x[0] = \{\text{"hello"}, \text{"world"}\} \]
  \[ z = \text{list}(x) \]
  \[ x[2] = 20 \quad \# z does not change \]
  \[ x[0][0] = \text{"HELLO"} \quad \# z changes \]

• \text{list} constructor does not perform a \textit{deep} copy
• \textit{Deep} copy: changes made to copy of object do not reflect in original object
• Can use Recursion for \textit{deep} copy of nested list

Mutability is Tricky

• All above scenarios can often lead to buggy code.
• Understanding the basics of mutability really helps in debugging your code.

However, mutability allow data objects to change state over time.

Is vs ==?

• == only compares values
• "is" compares whether two variables actually point to the same list
• Example:
  \[ x = [1, 2, 3, 4] \]
  \[ y1 = x \]
  \[ y2 = \text{list}(x) \]
  \[ \text{print} (y1 == x) \]
  \[ \text{print} (y2 == x) \]
  \[ \text{print} (y1 is x) \]
  \[ \text{print} (y2 is x) \]
Mutability and Nonlocal

- Consider the following example:

```python
def outer():
    x = 5
    def inner():
        x = 6 # Will this change the value of the outer x?
        return inner()
    return inner()
outer()
```

- `inner()` does not modify the outer variable; it will create a new local variable.
- However!!

```python
def outer():
    x = 5
    def inner():
        x[0] = 6 # Will this change outer x?
        return inner()
    return inner()
outer()
```

Mutability and Nonlocal

- Mutable objects can change inside `inner()`
- To change immutable objects inside `inner()`, we must use the `nonlocal` keyword:

```python
def outer():
    x = 5
    def inner():
        nonlocal x
        x = 6
        return inner()
    return inner()
outer()
```

- `Nonlocal` will not allow you to change global variables in this manner.
- To do this, you must use the global keyword:

```python
def outer():
    x = 5
    def inner():
        global x
        x = 6
        return inner()
    return inner()
outer()
```

Why Nonlocal?

- Create a Function with local state:

```python
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw
wd = make_withdraw(20)
w2 = make_withdraw(7)
print(wd(15))
print(wd(6))
```

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).

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):
    try:
        quotient = x/y
    except ZeroDivisionError:
        print('Can’t divide by zero!')
    return quotient
result = safe_divide(5,0)
print('Result is:', result)
```

- Can’t divide by zero!
  - Result is: Error
More on Exceptions

- Allows modular programs
- More exceptions types:
  [https://tinyurl.com/nI2yhry](https://tinyurl.com/nI2yhry)

- In general, a significant portion of code is exception handling.
- Some use the 80/20 rule: 20% of the code is for actual application, 80% is exception handling.