Lecture 08, 07 September 2023

Mutable and immutable values

- Lists are dictionaries are mutable
- All other values are immutable (numbers, booleans, strings, tuples)

- *immutable value* : If `x` holds an immutable value, `y = x` copies the value to `y`
- *mutable value* : If `l1` holds a mutable value, `l2 = l1` makes both names point to the same value

Functions and parameters

- Pass a mutable value, then it can updated in the function
- Immutable values will be copied

In [1]:
```
def mycopy(m,n):
    m = n
```

In [2]:
```
a = 5
b = 7
mycopy(a,b)
```

In [3]:
```
a, b
```

Out[3]:
```
(5, 7)
```

In [4]:
```
l1 = [1,2]
l2 = [3,4]
mycopy(11,l2)
```

In [5]:
```
l1, l2
```

Out[5]:
```
([1, 2], [3, 4])
```

In [6]:
```
def mycopylist(m,n):
    print(type(m))
    m[0] = n[-1]
```

In [7]:
```
l1 = [1,2]
l2 = [3,4]
mycopylist(l1,l2)
```

```
<class 'list'>
```

In [8]:
```
l1, l2
```

Out[8]:
```
([4, 2], [3, 4])
```

In [9]:
```
def myappend(l,v):
    l.append(v)
```

In [10]:
```
myappend(l2,5)
```

In [11]:
```
l2
```

Out[11]:
```
[3, 4, 5]
```

In [12]:
```
def myappend2(l,v):
    l = l + [v]
```

In [13]:
```
myappend2(l2,6)
```

In [14]:
```
l2
```

Out[14]:
```
[3, 4, 5]
```

Mutability and functions

It is useful to be able to update a list inside a function --- e.g. sorting it

- Built in list functions update in place
- `l.append(v)` -> in place version of `l = l+[v]`
- `l.extend(l1)` -> in place version of `l = l + l1`

In [ ]:

```python

```

Lists, arrays, dictionaries: implementation details

- What are the salient differences?
- How are they stored?
- What is the impact on performance?

Arrays

- Contiguous block of memory
- Typically size is declared in advance, all values are uniform
- `a[0]` points to first memory location in the allocated block
- Locate `a[i]` in memory using index arithmetic
  - Skip `i` blocks of memory, each block's size determined by value stored in array
- **Random access** -- accessing the value at `a[i]` does not depend on `i`
- Useful for procedures like sorting, where we need to swap out of order values `a[i]` and `a[j]`
  - `a[i], a[j] = a[j], a[i]`
  - Cost of such a swap is constant, independent of where the elements to be swapped are in the array
- Inserting or deleting a value is expensive
- Need to shift elements right or left, respectively, depending on the location of the modification

Lists

- Each location is a *cell*, consisiting of a value and a link to the next cell
  - Think of a list as a train, made up of a linked sequence of cells
- The name of the list `l` gives us access to `l[0]` , the first cell
- To reach cell `l[i]` , we must traverse the links from `l[0]` to `l[1]` to `l[2]` … to `l[i-1]` to `l[i]`
  - Takes time proportional to `i`
- Cost of swapping `l[i]` and `l[j]` varies, depending on values `i` and `j`
- On the other hand, if we are already at `l[i]` modifying the list is easy
  - *Insert* - create a new cell and reroute the links
  - *Delete* - bypass the deleted cell by rerouting the links
- Each insert/delete requires a fixed amount of local "plumbing", independent of where in the list it is performed

Dictionaries

- Values are stored in a fixed block of size m
- Keys are mapped to $\{0, 1, \ldots, m-1\}$
- Hash function $h : K \rightarrow S$ maps a *large* set of keys K to a *small* range S
- Simple hash function: interpret $k \in K$ as a bit sequence representing a number n_k in binary, and compute $n_k \mod m$, where $|S| = m$
- Mismatch in sizes means that there will be *collisions* -- $k_1 \neq k_2$, but $h(k_1) = h(k_2)$
- A good hash function maps keys "randomly" to minimize collisions
- Hash can be used as a *signature* of authenticity
  - Modifying k slightly will drastically alter $h(k)$
  - No easy way to reverse engineer a k' to map to a given $h(k)$
  - Use to check that large files have not been tampered with in transit, either due to network errors or malicious intervention
- Dictionary uses a hash function to map key values to storage locations
- Lookup requires computing $h(k)$ which takes roughly the same time for any k
  - Compare with computing the offset `a[i]` for any index `i` in an array
- Collisions are inevitable, different mechanisms to manage this, which we will not discuss now
- Effectively, a dictionary combines flexibility with random access

Lists in Python

- Flexible size, allow inserting/deleting elements in between
- However, implementation is an array, rather than a list
- Initially allocate a block of storage to the list
- When storage runs out, double the allocation
- `l.append(x)` is efficient, moves the right end of the list one position forward within the array
- `l.insert(0,x)` inserts a value at the start, expensive because it requires shifting all the elements by 1
- We will run experiments to validate these claims

Measuring execution time

- Call `time.perf_counter()`
- Actual return value is meaningless, but difference between two calls measures time in seconds

In [15]:

```python
import time
```

- 10^7 appends to an empty Python list

In [16]:
```
start = time.perf_counter()
l = []
for i in range(10000000):
    l.append(i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
3.2351508629944874
```

- Doubling the work approximately doubles the time, linear

In [17]:
```
start = time.perf_counter()
l = []
for i in range(20000000):
    l.append(i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
5.7539322239899775
```

- 10^5 inserts at the beginning of a Python list

In [18]:
```
start = time.perf_counter()
l = []
for i in range(100000):
    l.insert(0,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
5.2932833380036755
```

- Doubling and tripling the work multiplies the time by 4 and 9, respectively, so quadratic

In [19]:
```
start = time.perf_counter()
l = []
for i in range(200000):
    l.insert(0,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
17.236067117002676
```

In [20]:
```
start = time.perf_counter()
l = []
for i in range(300000):
    l.insert(0,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
44.08497351700498
```

- Creating 10^7 entries in an empty dictionary

In [21]:
```
start = time.perf_counter()
d = {}
for i in range(10000000,0,-1):
    d[i] = i
elapsed = time.perf_counter() - start
print(elapsed)
```

```
4.113557312011835
```

- Doubling the operations, doubles the time, so linear
- Dictionaries are effectively random access

In [22]:
```
start = time.perf_counter()
d = {}
for i in range(20000000,0,-1):
    d[i] = i
elapsed = time.perf_counter() - start
print(elapsed)
```

```
9.394316827994771
```