Lecture 8, 18 Oct 2021

Dictionaries

Accumulating values

- We have a list of pairs (name,marks) of marks in assignments of students in a course
- We want to report the total marks of each student
- Create a dictionary `total` whose keys are names and whose values are total marks for that name
- How would we do this?

```
marklist = [("abha",75),("bunty",58),("abha",86),("chitra",77),("bun

total = {}

for markpair in marklist:
  name = markpair[0]
  marks = markpair[1]
  # add marks to total[name], only if tota[name] already exist, othe
  if name in total.keys():  # check if a key exists already
    total[name] = total[name] + marks
  else:
    total[name] = marks

print(total)

```

```
{'abha': 161, 'bunty': 150, 'chitra': 77}
```

Representing sets

- Maintain a set X (from a universe U)
- Representing sets using functions
  - A subset $X \subseteq U$ is the same as a function $X : U \to \{\text{True}, \text{False}\}$
  - Say, $U = \{0, 1, \ldots, 999\}$, P = primes in U
  - $P = \{2, 3, 5, 7, \ldots, 997\}$
  - $P : \{0, 1, \ldots, 999\} \to \{\text{True}, \text{False}\}$
- Create a dictionary whose keys are those values x for which $P(x) = \text{True}$
  - `primes = {}`
  - `primes[2] = True`
  - `primes[3] = True`
  - …
  - `primes[997] = True`

- The set is implicitly the collection of keys of the dictionary
    - Can also explicitly add `primes[0] = False`, `primes[1] = False`, ..., but this is redundant
- **Exercise**: If `d1` and `d2` both represent sets over U, how do we compute `d1 ∪ d2`, `d1 ∩ d2`, $U \setminus$ `d1` (complement of `d1` wrt U)?

```
def factors(n):
  fl = []
  for i in range(1,n+1):
    if n%i == 0:
      fl.append(i)
  return(fl)

def prime(n):
  return(factors(n) == [1,n])

primes = {}
composites = {}
evens = {}
odds = {}

for i in range(50):
  if prime(i):
    primes[i] = True
  else:
    composites[i] = True
  if i%2·==·0:
    evens[i] = True
  else:
    odds[i] = True
```

```
composites
```

```
{0: True,
 1: True,
 4: True,
 6: True,
 8: True,
 9: True,
 10: True,
 12: True,
 14: True,
 15: True,
 16: True,
 18: True,
 20: True,
 21: True,
 22: True,
 24: True,
 25: True,
 26: True,
 27: True,
 28: True,
 30: True,
```

```
 32: True,
 33: True,
 34: True,
 35: True,
 36: True,
 38: True,
 39: True,
 40: True,
 42: True,
 44: True,
 45: True,
 46: True,
 48: True,
 49: True}
```

```
def setunion(s1,s2):
  newset = {}
  for k in s1.keys():
    newset[k] = True
  for k in s2.keys():
    newset[k] = True
  return(newset)

def setintersect(s1,s2):
  newset = {}
  for k in s1.keys():
    if k in s2.keys():   # Does not involve scanning all of s2 for s
                         # Different from "if y in l2"
      newset[k] = True
  return(newset)
```

```
print(setunion(primes,composites))
```

- Note that keys of `newset` are listed in the order they were added

```
print(setunion(composites,primes))
```

```
: True, 19: True, 23: True, 29: True, 31: True, 37: True, 41: True, 43: True, 47: True}
```

```
print(setintersect(odds,composites))
```

```
{1: True, 9: True, 15: True, 21: True, 25: True, 27: True, 33: True, 35: True, 39: True
```

- Mathematically, $S_1 \cup S_2 = S_2 \cup S_1$ -- set union is commutative
- In our dictionary representation, the internal structure differs
- However, if only use the dictionary in the context of set operations, there is no difference in the functionality
- Separating the *interface* from the *implementation* -- we will return to this idea often

Compare with list intersection

- To compute elements common to two lists we wrote

```
commonlist = []
for x in l1:
if x in l2:
  commonlist.append(x)
```

The check `if x in l2` requires a linear scan through `l2`

- For dictionaries, the corresponding code to check intersection of keys is

```
commonkeys = []
for k in d1.keys():
if k in d2.keys():
  commonkeys.append(k)
```

Superficially, these look similar, but the check `if k in d2.keys()` does not involve scanning a list of keys. As we shall see, we can quickly compute whether `k` is a key in `d2` or not

Deleting a key

- Use the function `del()`

```
d = {}
d["a"] = True
d["b"] = True
print(d)
# Now, remove the key "a"
del(d["a"])
print(d)
```

```
{'a': True, 'b': True}
{'b': True}
```

- More generally, `del` "unassigns" a value, makes a name undefined

```
x = 7
y = 8
z = x+y
b
del(x)
z = x+y
```

```
7 8 15
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-112-0bc16f0ee904> in <module>()
      4 print(x,y,z)
      5 del(x)
----> 6 z = x+y
```

- What about lists?
- `del(l[i])` deletes the value at position `i`
- This gap is filled by moving values beyond `i` to the left by 1
- To delete a segment, reassing a slice to `[]`

```
l = list(range(10))
print(l)ma
del(l[5])
print(l)
l[2:5] = []
print(l)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 6, 7, 8, 9]
[0, 1, 6, 7, 8, 9]
```

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 $n_k \bmod m$, 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

0s completed at 2:29 PM