Lecture 7, 30 August 2022

Operating on dictionaries

- How do we run through all entries in a dictionary - the equivalent of `for x in l`?
- `d.keys()`, `d.values()` generate sequences corresponding to the keys and values of `d`, respectively
- Like `range()` these are not directly lists, use `list(d.keys())` if you want a list

In [1]:
```
names = ["Abha","Bunty"]
bdays = {"Abha":"03-05-2001", "Bunty":"17-10-1999"}
bdays["Chitra"] = "13-09-2000"
```

In [2]:
```
bdays.keys()  # Not quite a list, a bit like range()
```

Out[2]:
```
dict_keys(['Abha', 'Bunty', 'Chitra'])
```

In [3]:
```
for name in bdays.keys():
  print(name)
```

```
Abha
Bunty
Chitra
```

- Shortcut, can omit `.keys()` when iterating

In [4]:
```
for name in bdays:
  print(name)
```

```
Abha
Bunty
Chitra
```

In [5]:
```
namelist = list(bdays.keys())
namelist
```

Out[5]:
```
['Abha', 'Bunty', 'Chitra']
```

- Similarly, list of values in a dictionary

In [6]:
```
list(bdays.values())
```

Out[6]:
```
['03-05-2001', '17-10-1999', '13-09-2000']
```

- In what order does `d.keys()` list the keys?
- In theory, this order is arbitrary and you should not make any assumptions
- In practice, from some recent version of Python (3.6?) keys are listed in the order added
- If dictionary keys are of the same type, use `sorted(d.keys())` to get them in sorted

In [7]:
```
d = {}
d['a'] = 7
d['c'] = 9
d['b'] = 8
```

In [8]:
```
list(d.keys()), sorted(d.keys())
```

Out[8]:
```
(['a', 'c', 'b'], ['a', 'b', 'c'])
```

- Noted that `sorted` implicitly produces a list, so no need to say `list(sorted(..))`

In [9]:
```
for name in sorted(d.keys()):
  print(d[name])
```

```
7
8
9
```

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?

In [10]:
```
marklist = [("abha",75),("bunty",58),("abha",86),("chitra",77),("bunty",92)]
total = {}
for markpair in marklist:
  name = markpair[0]
  marks = markpair[1]
  # add marks to total[name], only if tota[name] already exist, otherwise create a fresh entry
  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}
```

- Can accumulate other values -- for instance, the list of marks for each student

In [11]:
```
marklist = [("abha",75),("bunty",58),("abha",86),("chitra",77),("bunty",92)]
total = {}
for markpair in marklist:
  name = markpair[0]
  marks = markpair[1]
  # add marks to total[name], only if tota[name] already exist, otherwise create a fresh entry
  if name in total.keys():  # check if a key exists already
    total[name] = total[name] + [marks]
  else:
    total[name] = [marks]
print(total)
```

```
{'abha': [75, 86], 'bunty': [58, 92], '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 $X : U \rightarrow \{\text{True}, \text{False}\}$
  - Say, $U = \{0, 1, \ldots, 999\}$, P = primes in U
  - $P = \{2, 3, 5, 7, \ldots, 997\}$
  - $P : \{0, 1, \ldots, 999\} \rightarrow \{\text{True}, \text{False}\}$
- 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` $\cup$ `d2`, `d1` $\cap$ `d2`, $U \setminus$ `d1` (complement of `d1` wrt U)?

In [12]:
```
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
```

In [13]:
```
composites
```

Out[13]:
```
{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}
```

In [14]:
```
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 s2[k]
                         # Different from "if y in l2"
      newset[k] = True
  return(newset)
```

In [15]:
```
print(setunion(primes,composites))
```

```
{2: True, 3: True, 5: True, 7: True, 11: True, 13: True, 17: True, 19: True, 23: True, 29: True, 31: Tru
e, 37: True, 41: True, 43: True, 47: True, 0: True, 1: True, 4: True, 6: True, 8: True, 9: True, 10: Tru
e, 12: True, 14: True, 15: True, 16: True, 18: True, 20: True, 21: True, 22: True, 24: True, 25: True, 2
6: True, 27: True, 28: True, 30: True, 32: True, 33: True, 34: True, 35: True, 36: True, 38: True, 39: Tr
ue, 40: True, 42: True, 44: True, 45: True, 46: True, 48: True, 49: True}
```

- Note that keys of `newset` are listed in the order they were added
- The order changes if we reverse the arguments

In [16]:
```
print(setunion(composites,primes))
```

```
{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: T
rue, 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, 2: True, 3: True, 5: True, 7: True, 11: True, 13: True, 17: True, 19: True,
23: True, 29: True, 31: True, 37: True, 41: True, 43: True, 47: 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

- We can print the keys in sorted order

In [17]:
```
print(sorted(setunion(primes,composites)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 2
8, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
```

- Intersection works similarly -- note that the output preserves the order of the first set

In [18]:
```
print(setintersect(odds,composites))
```

```
{1: True, 9: True, 15: True, 21: True, 25: True, 27: True, 33: True, 35: True, 39: True, 45: True, 49: Tr
ue}
```

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()`

In [19]:
```
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

In [20]:
```
x = 7
y = 8
z = x+y
print(z,x,y)
```

```
15 7 8
```

In [21]:
```
x = 7
y = 8
del(x)
z = x+y
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_297213/1060327618.py in <module>
      2 y = 8
      3 del(x)
----> 4 z = x+y

NameError: name 'x' is not defined
```

- 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 `[]`

In [22]:
```
l = list(range(10))
print(l)
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 $n_k \mod m$, 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