FUNCTIONS AS OBJECTS, DICTIONARIES

FUNCTIONS AS OBJECTS

- functions are **first class objects:**
  - have types
  - can be elements of data structures like lists
  - can appear in expressions
    - as part of an assignment statement
    - as an argument to a function!!
- particularly useful to use functions as arguments when coupled with lists
  - aka **higher order programming**

EXAMPLE

```
def applyToEach(L, f):
    """assumes L is a list, f a function
       mutates L by replacing each element,
       e, of L by f(e)"""
    for i in range(len(L)):
        L[i] = f(L[i])
```

EXAMPLE

```
def applyToEach(L, f):
    for i in range(len(L)):
        L[i] = f(L[i])
```

```
L = [1, -2, 3.4]

applyToEach(L, abs)

applyToEach(L, int)

applyToEach(L, fact)

applyToEach(L, fib)
```

EXAMPLE

```
def applyToEach(L, f):
    for i in range(len(L)):
        L[i] = f(L[i])
```

```
L = [1, -2, 3.4]
```

```
applyToEach(L, abs)
```

```
[1, 2, 3.4]
```

```
applyToEach(L, int)

applyToEach(L, fact)

applyToEach(L, fib)
```

EXAMPLE

```
def applyToEach(L, f):
    for i in range(len(L)):
        L[i] = f(L[i])
```

```
L = [1, -2, 3.4]
```

```
applyToEach(L, abs)
```

```
[1, 2, 3.4]
```

```
applyToEach(L, int)
```

```
[1, 2, 3]
```

```
applyToEach(L, fact)
```

```
applyToEach(L, fib)
```

EXAMPLE

```
def applyToEach(L, f):
    for i in range(len(L)):
        L[i] = f(L[i])
```

```
L = [1, -2, 3.4]
```

```
applyToEach(L, abs)
```

```
[1, 2, 3.4]
```

```
applyToEach(L, int)
```

```
[1, 2, 3]
```

```
applyToEach(L, fact)
```

```
[1, 2, 6]
```

```
applyToEach(L, fib)
```

EXAMPLE

```
def applyToEach(L, f):
    for i in range(len(L)):
        L[i] = f(L[i])
```

```
L = [1, -2, 3.4]
applyToEach(L, abs)
```

```
[1, 2, 3.4]
applyToEach(L, int)
```

```
[1, 2, 3]
applyToEach(L, fact)
```

```
[1, 2, 6]
applyToEach(L, fib)
```

```
[1, 2, 13]
```

LISTS OF FUNCTIONS

```
def applyFuns(L, x):
    for f in L:
        print(f(x))

applyFuns([abs, int, fact, fib], 4)
4
4
24
5
```

GENERALIZATION OF HOPS

- Python provides a general purpose HOP, `map`
- simple form – a unary function and a collection of suitable arguments
  - `map(abs, [1, -2, 3, -4])`
- produces an 'iterable', so need to walk down it

```
for elt in map(abs, [1, -2, 3, -4]):
    print(elt)
```

```
[1, 2, 3, 4]
```

remember range?

- general form – an n-ary function and n collections of arguments
  - `L1 = [1, 28, 36]`
  - `L2 = [2, 57, 9]`

```
for elt in map(min, L1, L2):
    print(elt)
```

```
[1, 28, 9]
```

STRINGS, TUPLES, RANGES, LISTS

- Common operations
  - `seq[i]` → i^{th} element of sequence
  - `len(seq)` → length of sequence
  - `seq1 + seq2` → concatenation of sequences (not range)
  - `n*seq` → sequence that repeats `seq` n times (not range)
  - `seq[start:end]` → slice of sequence
  - `e in seq` → `True` if e contained in sequence
  - `e not in seq` → `True` if e contained in sequence
  - `for e in seq` → iterates over elements of sequence

PROPERTIES

Type	Type of elements	Examples of literals	Mutable
str	characters	' ', 'a', 'abc'	No
tuple	any type	(), (3,), ('abc', 4)	No
range	integers	range(10), range(1,10,2)	No
list	any type	[], [3], ['abc', 4]	Yes

DICTIONARIES

HOW TO STORE STUDENT INFO

- so far, can store using separate lists for every info

```
names = ['Ana', 'John', 'Denise', 'Katy']
grade = ['B', 'A+', 'A', 'A']
course = [2.00, 6.0001, 20.002, 9.01]
```

- a **separate list** for each item
- each list must have the **same length**
- info stored across lists at **same index**, each index refers to info for a different person

HOW TO UPDATE/RETRIEVE STUDENT INFO

```
def get_grade(student, name_list, grade_list, course_list):
    i = name_list.index(student)
    grade = grade_list[i]
    course = course_list[i]
    return (course, grade)
```

- **messy** if have a lot of different info to keep track of
- must maintain **many lists** and pass them as arguments
- must **always index** using integers
- must remember to change multiple lists

A BETTER AND CLEANER WAY – A DICTIONARY

- nice to **index item of interest directly** (not always int)
- nice to use **one data structure**, no separate lists

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
…	…

index element

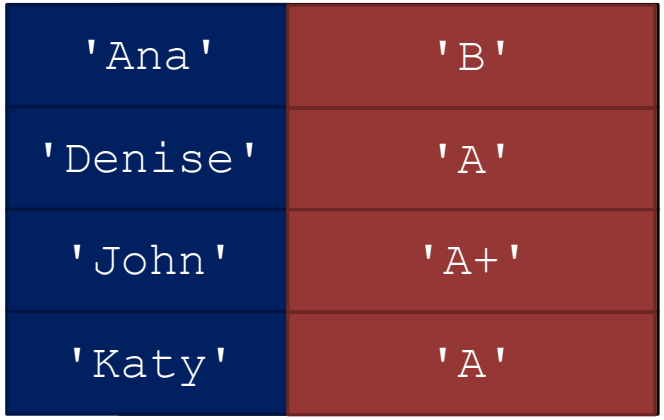
A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
…	…

custom index by label element

A PYTHON DICTIONARY

- store pairs of data
  - key
  - value



empty dictionary

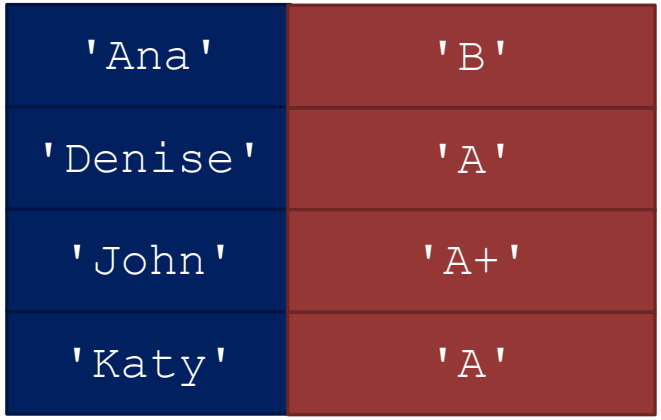
```
my_dict = {}
```

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

key1 val1 key2 val2 key3 val3 key4 val4

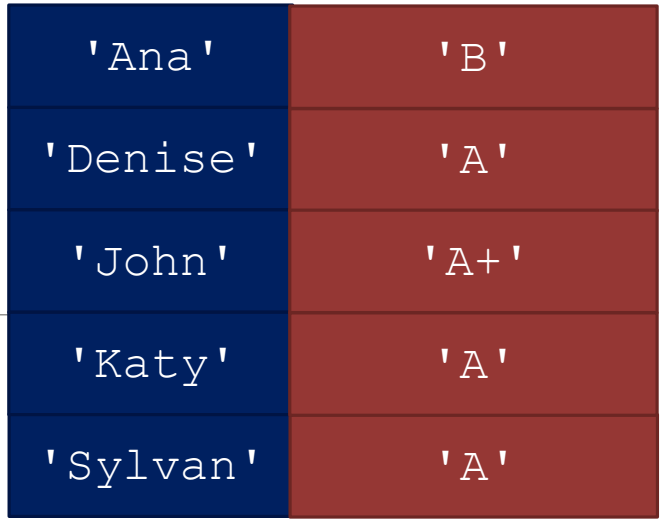
DICTIONARY LOOKUP

- similar to indexing into a list
- **looks up** the **key**
- **returns** the **value** associated with the key
- if key isn't found, get an error




```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
grades['John']      → evaluates to 'A+'
grades['Sylvan']    → gives a KeyError
```

DICTIONARY OPERATIONS




```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- **add** an entry

```
grades['Sylvan'] = 'A'
```

- **test** if key in dictionary

```
'John' in grades        → returns True
'Daniel' in grades      → returns False
```

- **delete** entry

```
del(grades['Ana'])
```

DICTIONARY OPERATIONS

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- get an **iterable that acts like a tuple of all keys**

  no guaranteed order

  `grades.keys()` → returns `['Denise','Katy','John','Ana']`

- get an **iterable that acts like a tuple of all values**

  `grades.values()` → returns `['A', 'A', 'A+', 'B']`

  no guaranteed order

DICTIONARY KEYS and VALUES

- values
  - any type (**immutable and mutable**)
  - can be **duplicates**
  - dictionary values can be lists, even other dictionaries!
- keys
  - must be **unique**
  - **immutable** type (`int`, `float`, `string`, `tuple`, `bool`)
    - actually need an object that is **hashable,** but think of as immutable as all immutable types are hashable
  - careful with `float` type as a key
- **no order** to keys or values!

```
d = {4:{1:0}, (1,3):"twelve", 'const':[3.14,2.7,8.44]}
```

`list` vs `dict`

- **ordered** sequence of elements
- look up elements by an integer index
- indices have an **order**
- index is an **integer**

- **matches** "keys" to "values"
- look up one item by another item
- **no order** is guaranteed
- key can be any **immutable** type

EXAMPLE: 3 FUNCTIONS TO ANALYZE SONG LYRICS

1) create a **frequency dictionary** mapping `str:int`

2) find **word that occurs the most** and how many times
   - use a list, in case there is more than one word
   - return a tuple `(list,int)` for (words_list, highest_freq)

3) find the **words that occur at least X times**
   - let user choose "at least X times", so allow as parameterS
   - return a list of tuples, each tuple is a `(list, int)` containing the list of words ordered by their frequency
   - IDEA: From song dictionary, find most frequent word. Delete most common word. Repeat. It works because you are mutating the song dictionary.

CREATING A DICTIONARY

```
def lyrics_to_frequencies(lyrics):
    myDict = {}
    for word in lyrics:
        if word in myDict:
            myDict[word] += 1
        else:
            myDict[word] = 1
    return myDict
```

can iterate over list

can iterate over keys in dictionary

update value associated with key

USING THE DICTIONARY

```
def most_common_words(freqs):
    values = freqs.values()
    best = max(values)
    words = []
    for k in freqs:
        if freqs[k] == best:
            words.append(k)
    return (words, best)
```

this is an iterable, so can apply built-in function

can iterate over keys in dictionary

LEVERAGING DICTIONARY PROPERTIES

```
def words_often(freqs, minTimes):
    result = []
    done = False
    while not done:
        temp = most_common_words(freqs)
        if temp[1] >= minTimes:
            result.append(temp)
            for w in temp[0]:
                del(freqs[w])
        else:
            done = True
    return result


print(words_often(beatles, 5))
```

can directly mutate dictionary; makes it easier to iterate

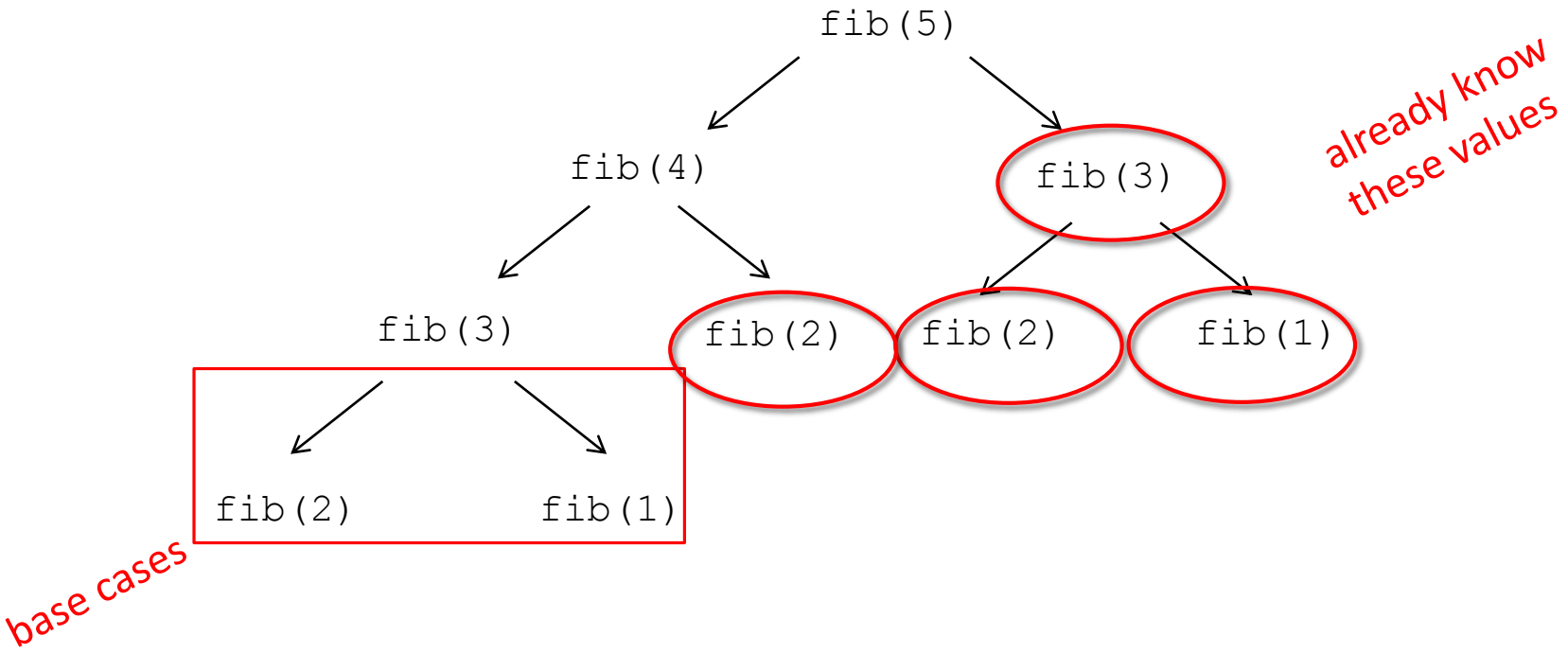
FIBONACCI RECURSIVE CODE

```
def fib(n):
    if n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return fib(n-1) + fib(n-2)
```

- two base cases
- calls itself twice
- this code is inefficient

INEFFICIENT FIBONACCI

`fib(n) = fib(n-1) + fib(n-2)`



- **recalculating** the same values many times!
- could keep **track** of already calculated values

FIBONACCI WITH A DICTIONARY

```
def fib_efficient(n, d):
    if n in d:
        return d[n]
    else:
        ans = fib_efficient(n-1, d) + fib_efficient(n-2, d)
        d[n] = ans
        return ans

d = {1:1, 2:2}
print(fib_efficient(6, d))
```

Method sometimes called "memoization"

- do a **lookup first** in case already calculated the value
- **modify dictionary** as progress through function calls

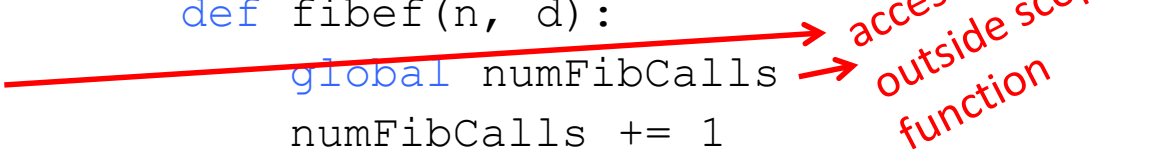
GLOBAL VARIABLES

- can be dangerous to use
  - breaks the scoping of variables by function call
  - allows for side effects of changing variable values in ways that affect other computation
- but can be convenient when want to keep track of information inside a function
- example – measuring how often `fib` and `fib_efficient` are called

TRACKING EFFICIENCY

```
def fib(n):
    global numFibCalls
    numFibCalls += 1
    if n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return fib(n-1)+fib(n-2)
```

```
def fibef(n, d):
    global numFibCalls
    numFibCalls += 1
    if n in d:
        return d[n]
    else:
        ans = fibef(n-1,d)+fibef(n-2,d)
        d[n] = ans
        return ans
```



TRACKING EFFICIENCY

```
numFibCalls = 0

print(fib(12))
print('function calls', numFibCalls)

numFibCalls = 0

d = {1:1, 2:2}
print(fib_efficient(12, d))
print('function calls', numFibCalls)
```