# 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!!

Numbers have a type.
They can be elements
of a data structure
and they can appear
inside of expressions.
Same thing with
strings.
It turns out functions
can as well

- particularly useful to use functions as arguments when coupled with lists
  - aka **higher order programming**

using functions, for example, inside
of lists to do things on lists.

# 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])
```

iterating down the length of the list.
for each index that goes into the list, get out the element of the list,
apply f to it, and then put that back in that spot inside of the list.
->
So I'm literally applying that function to each element of the list.
I'm also mutating the list as I do it

# 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

given a data structure,
give me back a version of that data structure-
- in fact, it's pointing to the same thing--
where I've done something to each element of the list.
In this case using a function.

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

apply a function to each element of list

```
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))
```

I could do it the other direction,
I could apply a list of functions L to a number x

for each element (function) of that list L, apply it, to the argument, x,
and print out the result

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

higher order procedure

# 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])`

takes the function that expects only one argument,
takes a collection, in this case a list of appropriate arguments,
and it literally creates a list where
it has applied that function to each element in turn

- produces an 'iterable', so need to walk down it

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

map gives me back a structure
that's going to act like a list, but it's something
that I have to walk down, iterate over, to get back out

remember
range?

collection of functions that
expect more than one, n, arguments

- 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]
```

take the first element of each list,
apply that function to it, take the second element
of each list, apply that function to it,
and generate for us something that
has the result of doing that processing

# STRINGS, TUPLES, RANGES, LISTS

compound data structures
non-scalar (have internal structure that can be accessed) data structures,
ordered (sequence) collections of elements,
collecting data together into a structure

- 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 not 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

problem I've got is that the association isn't
captured in a common place

# 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

linear structure

**A list**

| | |
|---|---|
| 0 | Elem 1 |
| 1 | Elem 2 |
| 2 | Elem 3 |
| 3 | Elem 4 |
| … | … |

index

set of integers starting at 0

element

associated with each indice, I have a different element

**A dictionary**

| | |
|---|---|
| Key 1 | Val 1 |
| Key 2 | Val 2 |
| Key 3 | Val 3 |
| Key 4 | Val 4 |
| … | … |

custom index by label

element

give me the element associated with this key, call the indices keys, and use them as labels that tell me where to find the element inside of dict

# A PYTHON DICTIONARY

- store pairs of data
  - key
  - value

| key | value |
|---|---|
| 'Ana' | 'B' |
| 'Denise' | 'A' |
| 'John' | 'A+' |
| 'Katy' | 'A' |

custom index by label

element

empty dictionary

They're not going to be in any particular order, other than how I happen to put them into the 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

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

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

`grades['John']` → evaluates to `'A+'` give me back the value associated with that key

`grades['Sylvan']` → gives a `KeyError`

# DICTIONARY OPERATIONS

| | |
|---|---|
| 'Ana' | 'B' |
| 'Denise' | 'A' |
| 'John' | 'A+' |
| 'Katy' | 'A' |
| 'Sylvan' | 'A' |

```
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'}
```

order in which I put them in

it doesn't generate them

- 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

I get back things so that I could then iterate over--
like I could walk down all the collections of keys,
or I could walk down all the collections of values
doing something to them.

# DICTIONARY KEYS and VALUES

- values
  - any type (**immutable and mutable**)
  - can be **duplicates**
  - dictionary values can be lists, even other dictionaries!

ints, floats, strings, tuples
lists
functions.....

same value
to be associated with different keys

- 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

if the float has an accuracy issue,
I may not find the thing I wanted to associate with it

- **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

association of a word
and number of appearances

key value

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

2) find **word that occurs the most** and how many times

that occurs same number of 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
```

create an empty dictionary

iterate over the lyrics, getting each word out

f the word's already in the dictionary, increase the value associated with it by one

if the word is not in the dictionary, this is the first time I've seen the word. I'm going to set the value in the dictionary corresponding to that word to one

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)
```

Give me all the values in the dictionary. Ah, it's now a collection of integers. Just give me the maximum value.

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

empty list

can iterate over keys in dictionary

is word one of the most common words? (is value:int associated with key:word maximum of ints?)

(w, b) = most_common_words(beatles) returns a tuple

# 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))
```

set up an empty list initially

flag is initially set default to False
because I'm going to run through a loop here

True Flag

can directly mutate dictionary; makes it easier to iterate

I'm going to find most common words in the dictionary.
And if they occur more than the minimum number of times that I've set, I'm going to add them into my result

for everything in that word, I'm going to remove it from the dictionary

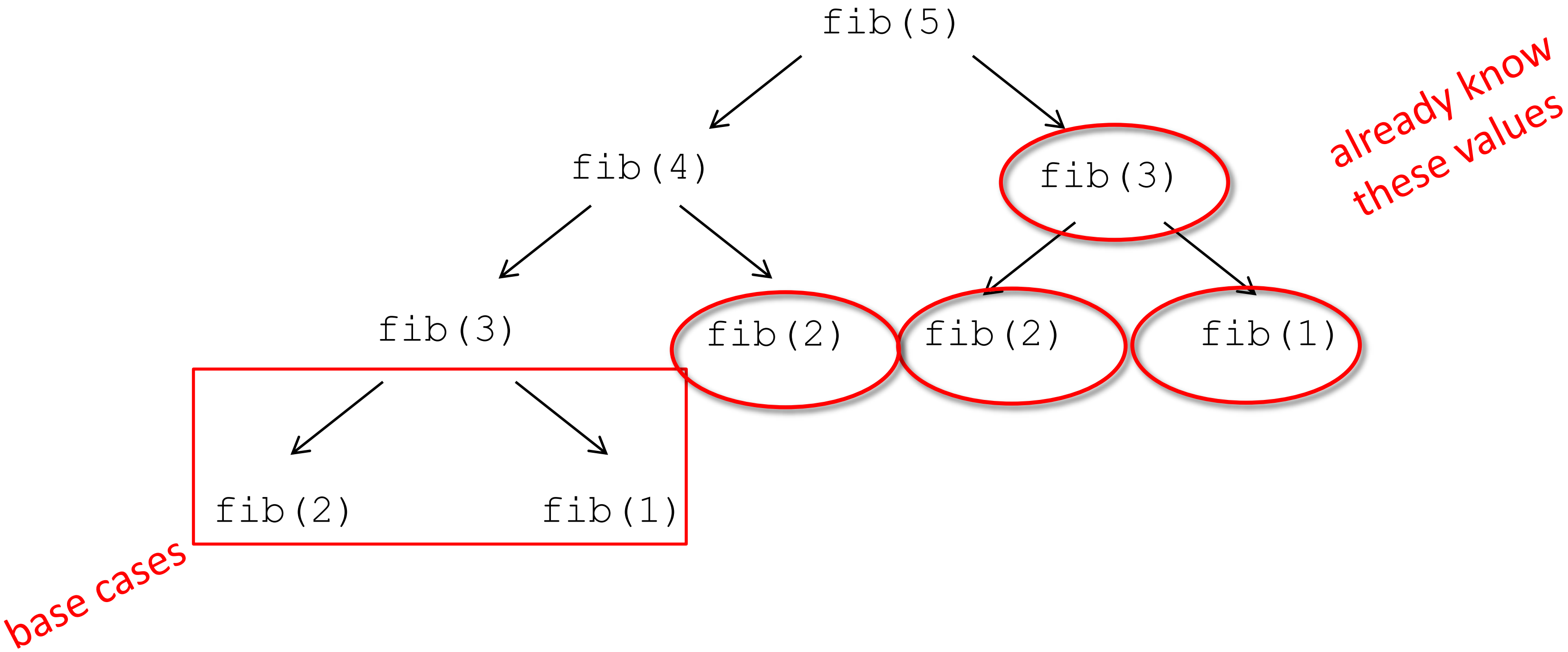
# FIBONACCI RECURSIVE CODE

```
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

arguments:
n for n-th Fibonacci number
d 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))
```

But if I've already done the work for n, just look n up as key and return n-th fibonacci number (ans) as associated value

Method sometimes called "memoization"

if I'm computing this Fibonacci number for the first time, I'll do the work as I normally would, and then I'm going to store it into the dictionary

I need to give the base case in the dictionary

- 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

global is a special term.
It says this variable name is
something that I can access
outside the scope of the function
(before this anything
inside the body of the function
was only accessible
within the call of the function itself)

```
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)
```

keep track of how often did I actually call this function

```
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
```

accessible from outside scope of function

# TRACKING EFFICIENCY

need to initialize that global variable outside.

```
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)
```