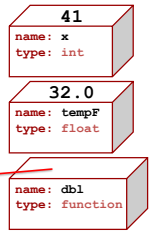
Remembering Data

```
x = 41

tempF = 32.0

def dbl(x):
    return 2*x
```



Just like numbers, strings, and lists, **functions are also data!**

Therefore, **functions can take other functions as input!**

Functions that take other functions as input or return functions are called **higher order functions**

```
def dbl(x):
    return 2*x

myList = [1, 2, dbl]

canWeDouble = myList[2]

canWeDouble(12)

>>> 24

>>> myList[2](3) # or simply
>>> 6
```

Mapping

- Map – The application of *one* specific operation to *each* element in a list
- Example: suppose we wanted to double the value of every number in a list, and output the new list?

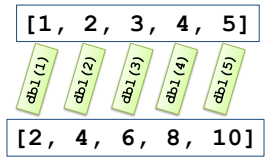
```
>>> dblList([1,2,3,4,5])
[2,4,6,8,10]
```

- Sure we can do it in list comprehension. But mapping makes it more genral.

A general approach…

- What if we could apply *any* arbitrary function to each element in a list to produce a new list?

```
def dbl(x):
    return 2*x
```



map!

- **`map(f,t)`** – A built-in function that applies any arbitrary function **`f`** to every element in **`t`**
  - **`t`** is any iterable object, list is an example

```
def dbl(x):
    return 2*x

lst = [1,2,3,4,5]

newLst = map(dbl,lst)
```

```
>>> newLst = map(dbl,lst)
>>> newLst
<map object at 0x17de3f0>

>>> list(newLst)
[2, 4, 6, 8, 10]
```

Why? **`map()`** returns an iterable object.

Map examples

```
def dbl(x):
    return 2*x
```

```
>>> list( map(dbl, [0,1,2,3,4,5]) )
[0, 2, 4, 6, 8, 10]

>>> list( map(dbl, 'test') )
['tt', 'ee', 'ss', 'tt']
```

```
def square(x):
    return x**2
```

```
>>> list( map(square, range(6)) )
[0, 1, 4, 9, 16, 25]
```

```
def isA(x):
    return x == 'a'
```

```
>>> list( map(isA, 'go away!') )
[False, False, False, True, False, True, False, False]
```

Map !

Without map

```
def dblList(lst):
    if lst == []:
        return lst
    else:
        return [lst[0]*2] + dblList(lst[1:])
```

With map!

```
def dbl(x):
    return x*2

def dblList(lst):
    return list( map(dbl, lst) )
```

Map v. Lists?

```
map( dbl, range(99))
 vs.
[ dbl(num) for num in range(99) ]
```

Map v. Lists?

```
map( dbl, range(9999999999999))
 vs.
[ dbl(num) for num in range(999999999999999) ]
```

Scalability!

Map binds (connects) function to data, it doesn't generate the list until referenced. List comprehension computes as listed.

```
>>> newLst = map(dbl,lst)
>>> newLst
<map object at 0x17de3f0>
```

```
>>> list(newLst)
[2, 4, 6, 8, 10]
```

Reducing lists

- **`reduce(f,t)`** – Applies **`f`** (a function of two arguments) *cumulatively* to the items of **`t`**
  - applied from left to right, so as to reduce the sequence to a single value
  - NOT a built-in function! Available in **`functools`** module
- Example:

```
def add(x,y):
    return x + y
from functools import reduce
>>> reduce(add, [1,2,3,4,5])
15
```

The process is 1+2, then 3+3, then 6+4, then 10+5.

NOTE: **`f`** *must* return the same type! Why?

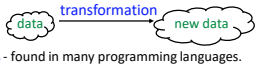
Filter

- **`filter(f,t)`** – constructs a list from those elements of t for which f returns True
  - applied from left to right
  - Example:

```
def is_vowel(x):
    return x in 'aeiou'

>>> x = filter(is_vowel, 'hello world')
>>> list(x)
>>> ['e', 'o', 'o']
```

Computations == Transformations



Common transformations - found in many programming languages.

Map - apply same action to every element in sequence.

[2, 7, 6, 4] —double→ [4, 14, 12, 8] (Remember: lists and strings are sequences.)

Filter - select certain items in a sequence by a predicate.

(A predicate is a function that returns True or False.)

[3, 2, 13, 17, 6] —isEven→ [2, 6]

Reduce - apply the same action between elements of a sequence.

reduce(add, [2, 3, 7, 4]) == (((2+3)+7)+4) == 16