???

Assignment 2 – due tomorrow!

```
except:

except: Exception
```

Mutating vs. non-mutating

9 bad tests – updated results

Not just correctness!

Midterm 2:
Recovery and Next Steps

Efficiency

Observation 1:

Running time (usually) depends on machine (and what else is running)

Count “steps”

Observation 2:

Definition of “step” doesn’t matter

```
for item in lst:
    print(item)
```

```
for item in lst:
    print(item)
return 5
```

```
for item in lst:
    x = 5
    y = item + 1
```

```
i = 0
while i < len(lst):
    print(lst[i])
    i += 1
```

Observation 3:

Running time depends on input size

```
def size(stack):
    count = 0
    temp = Stack()

    while not stack.is_empty():
        temp.push(stack.pop())
        count += 1

    while not temp.is_empty():
        stack.push(temp.pop())

    return count
```

Proportional to
n = size of stack

"linear time"

O(n)

Big-Oh Notation: O(__)

Ignore constants: 3n, n + 5, n, n – 100, 0.01n + 20

Focus on *asymptotic* behaviour

$O(\log n)$, $O(n)$, $O(n \log n)$
$O(n^2)$, $O(2^n)$

Sorting

Selection sort: $O(n^2)$

Mergesort: $O(n \log n)$

```
def remove_kth(stack, k):
    count = 0
    temp = Stack()

    while count < k:
        temp.push(stack.pop())
        count += 1

    kth = temp.pop()

    while not temp.is_empty():
        stack.push(temp.pop())

    return kth
```

Proportional to *k*

$O(k)$

```
def remove_kth(stack, k):
    count = 0
    temp = Stack()

    while count < k:
        temp.push(stack.pop())
        count += 1

    kth = temp.pop()

    while not temp.is_empty():
        stack.push(temp.pop())

    return kth
```

Worst case?

$O(n)$

Best case?

$O(1)$

O(1)

"constant time"

runtime doesn't depend on input size

Worst, Average, Best

Most list methods depend on length (and index)

(search, insert, delete, etc.)

```
def num_common(lst1, lst2):
    count = 0
    for x in lst1:
        for y in lst2:
            if x == y:
                count += 1
    return count
```

lst1 has length *n*
lst2 has length *m*

O(mn)

Most tree methods depend on size and/or height

Recurse on…

one subtree

multiple subtrees

Tree size vs. height

Worst case

Best case

Memory Model

“Data” is stored in two places:

stack and **heap**

(Note: special terms!)

Call stack

(keeping track of function calls)

Argument values

Local variables

Return address

Unique to each function *call**

*Except default values!!

Call stack is transient…

How is data kept between calls?

Heap

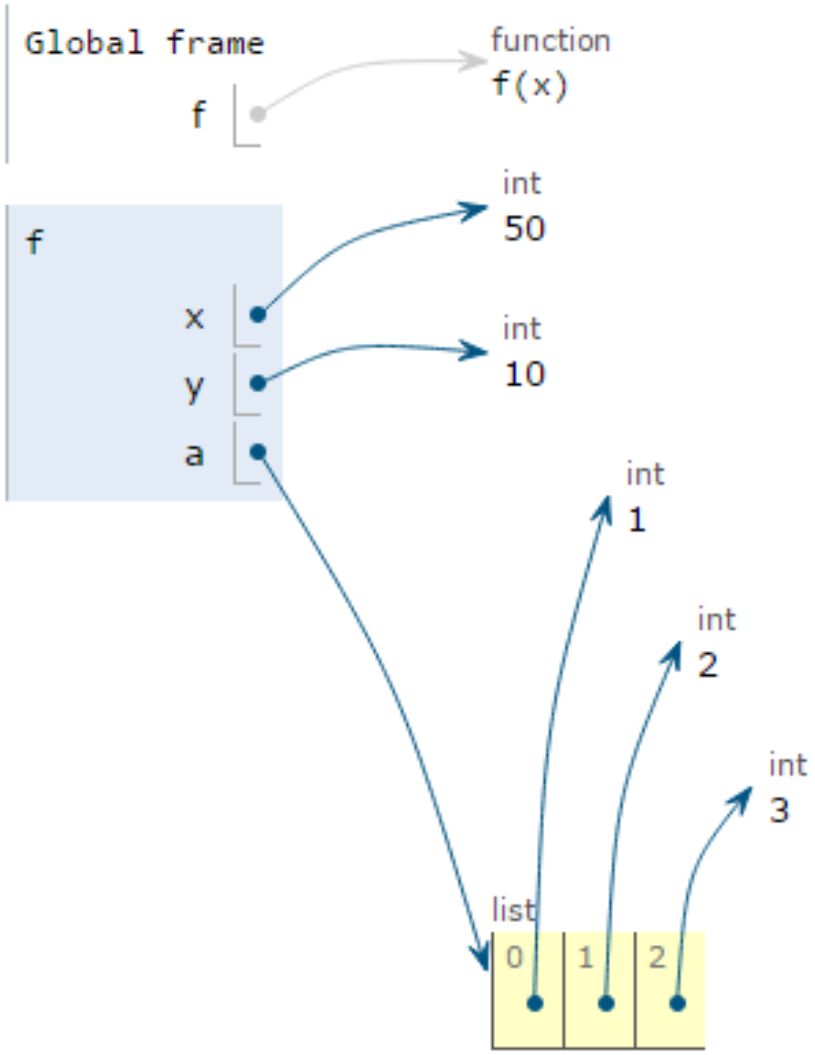
(where data lives)

Heap: memory available to program *not* used by stack

Numbers, lists, dictionaries, objects, classes, functions

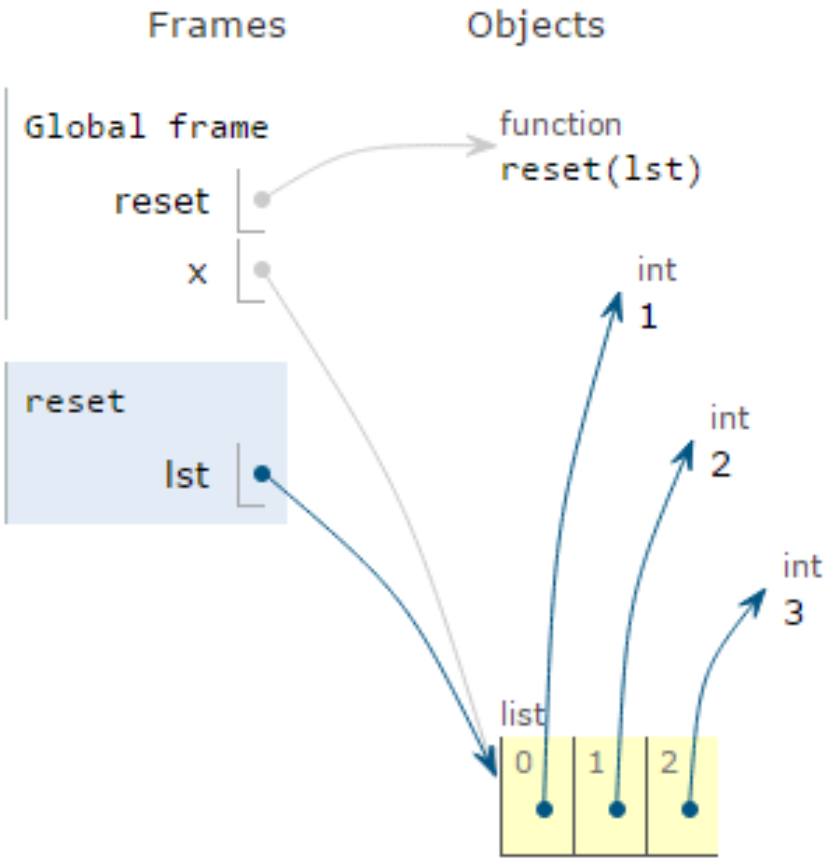
Variables store **references** to locations in the heap

```
def f(x):
    y = 10
    a = [1,2,3]

    print(x)
    return 10

f(50)
```



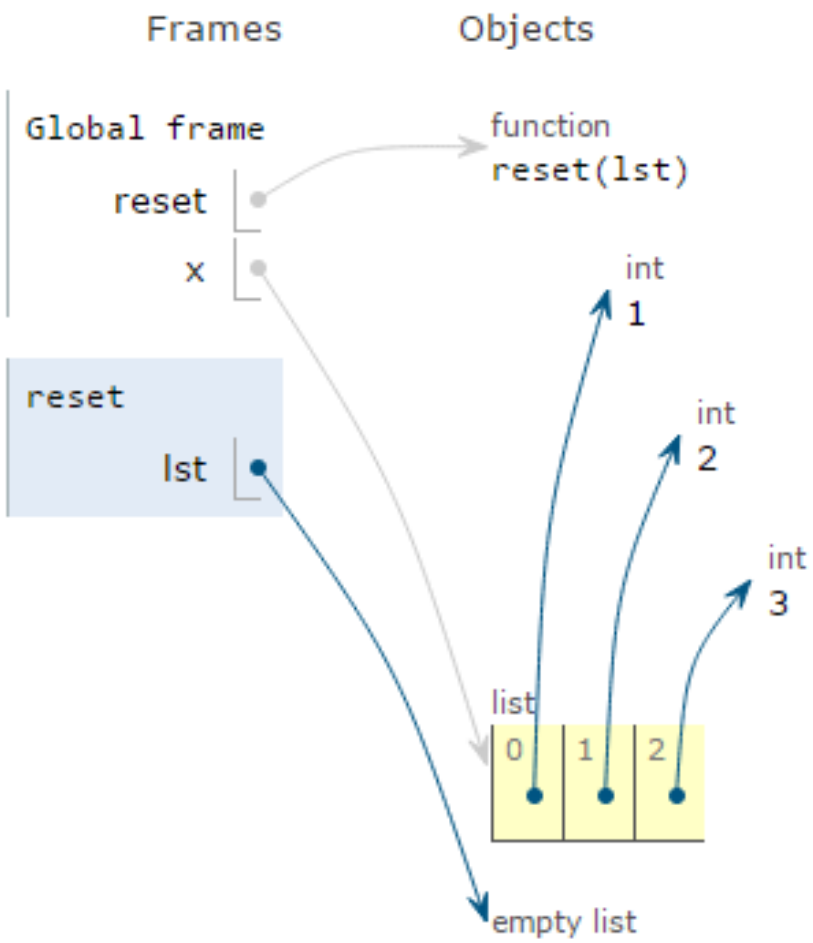
Parameters are **new** references

```
def reset(lst):
    lst = []

x = [1,2,3]
reset(x)
```



Parameters are **new** references

```
def reset(lst):
    lst = []

x = [1,2,3]
reset(x)
```



Morals

x = ___ changes a **reference**

x = y does **not** make a copy

Subtlety

Computer does a lot of work for you!

Garbage collection