SAMS
Programming - Section C

Week 3 - Lecture 1:
Lists
## Built-in Data Types

<table>
<thead>
<tr>
<th>Python name</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoneType</td>
<td>absence of value</td>
<td>None</td>
</tr>
<tr>
<td>bool (boolean)</td>
<td>Boolean values</td>
<td>True, False</td>
</tr>
<tr>
<td>int (integer)</td>
<td>integer values</td>
<td>$-2^{63}$ to $2^{63} - 1$</td>
</tr>
<tr>
<td>long</td>
<td>large integer values</td>
<td>all integers</td>
</tr>
<tr>
<td>float</td>
<td>fractional values</td>
<td>e.g. 3.14</td>
</tr>
<tr>
<td>complex</td>
<td>complex values</td>
<td>e.g. 1+5j</td>
</tr>
<tr>
<td>str (string)</td>
<td>text</td>
<td>e.g. “Hello World!”</td>
</tr>
<tr>
<td>list</td>
<td>a list of values</td>
<td>e.g. [2, “hi”, 3.14]</td>
</tr>
</tbody>
</table>
## String vs List

<table>
<thead>
<tr>
<th>String</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = “hw2-1 was hard”</code></td>
<td><code>a = [1, 3.14, “hi”, True]</code></td>
</tr>
<tr>
<td>A sequence (string) of characters.</td>
<td>A sequence of arbitrary objects.</td>
</tr>
<tr>
<td>Immutable</td>
<td>Mutable</td>
</tr>
<tr>
<td><code>s[0] = “H”</code></td>
<td><code>a[0] = 100</code></td>
</tr>
</tbody>
</table>
Lists: basic usage

```python
a = []  # creates an empty list
b = list()  # also creates an empty list
c = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
d = list(range(1, 11))  # also creates an empty list
e = [1, 3.14, None, True, "Hi", [1, 2, 3]]

for i in range(len(c)):
    print(c[i])

for item in e:
    print(item)

print(e[1:4])
e[2] = 0
d += [11]
print(d[::2])
```
Lists: basic usage

```python
print([1, 2, 3] + [4, 5, 6])  # [1, 2, 3, 4, 5, 6]
a = [0] * 5
print(a)  # [0, 0, 0, 0, 0]

if 1 in a:
    print("1 is in the list a.")

if 1 not in a:
    print("1 is not in the list a.")

b = [0, 0, 0, 0, 0, 0]

if a == b:
    print("a and b contain the same elements.")
```
Lists: built-in functions

```python
a = list(range(1, 11))
print(len(a))
print(min(a))
print(max(a))
print(sum(a))

a = [4, 5, 1, 3, 2, 8, 7, 6, 9, 10]

a = sorted(a)
print(a)  # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```
Lists: interesting example

```python
x = 1
y = x
x += 1
print(x, y)  # 2 1

x = [1, 2, 3]
y = x
x[0] = 4
print(x, y)  # [4, 2, 3] [4, 2, 3]
```
immutable vs mutable

Main memory (RAM)

Address

memory cell
immutable vs mutable

Immutable objects

\[ x = 5 \]
\[ y = 4 \]
\[ x = 1 \]
\[ y -= 2 \]
Immutable vs mutable

Immutable objects

\[ x = 5 \]
\[ y = 4 \]
\[ x = 1 \]
\[ y -= 2 \]
immutable vs mutable

Immutable objects

```
x = 5
y = 4
x = 1
y -= 2
```
Immutable vs mutable

**Immutable objects**

\[
\begin{align*}
x &= 5 \\
y &= 4 \\
x &= 1 \\
y &= 2
\end{align*}
\]

- Garbage

\[
\begin{array}{cccc}
x & \rightarrow & 2843 \\
& & 2844 \\
& & 2845 \\
\circ & & \circ & 5 \\
& & 2846 \\
& & 2847 \\
& & 2848 \\
& & 2849 \\
y & \rightarrow & 2850 \\
& & 1 \\
& & 2 \\
& & \vdots
\end{array}
\]
Immutable vs mutable

Immutable objects

```
x = 5
y = 4
x = 1
y -= 2
```

“Garbage is collected”
immutable vs mutable

**Immutable objects**

\[ x = 5 \]
\[ y = 4 \]
immutable vs mutable

**Immutable objects**

\[
x = 5 \\
y = 4 \\
x = 1
\]

\[
x \rightarrow 5 \\
y \rightarrow 4
\]
Immutable vs mutable

**Immutable objects**

\[
\begin{align*}
x &= 5 \\
y &= 4 \\
x &= 1
\end{align*}
\]
Immutable vs mutable

Immutable objects

\[
x = 5
\]
\[
y = 4
\]
\[
x = 1
\]
\[
y -= 2
\]
**Immutable vs Mutable**

**Immutable objects**

\[
\begin{align*}
x &= 5 \\
y &= 4 \\
x &= 1 \\
y &= 2
\end{align*}
\]
Immutable vs mutable

Immutable objects

\[ x = 5 \]

\[ x \rightarrow 5 \]
immutable vs mutable

Immutable objects

\[ x = 5 \]
\[ y = x \]
Immutable objects

\[ x = 5 \]
\[ y = x \]
\[ x += 1 \]

print(x, y)
immutable vs mutable

Immutable objects

\[ x = 5 \]
\[ y = x \]
\[ x += 1 \]

print(x, y) 6 5
Immutable vs mutable

Immutable objects

$$x = 5$$
$$y = x$$

In reality:

In practice:

if you change one, the other is not affected
So actually, a list is a sequence of references (variables)!
immutable vs mutable

**Mutable objects**

```
x = [1, 2, 3]
y = x
```

![Diagram](image)
Mutable objects

\[ x = [1, 2, 3] \]
\[ y = x \]
\[ x[0] = 4 \]
**Immutable vs Mutable**

**Mutable objects**

\[
x = [1, 2, 3] \]
\[
y = x \]
\[
x[0] = 4 \]
\[
\text{print}(y[0]) \quad 4
\]

\[
x \quad 4 \\
y \quad 1 \\
\quad 2 \\
\quad 3
\]

\[
x \quad \text{aliases}
\]

\[
x \quad \text{and} \quad y \quad \text{are} \quad \text{aliases}
\]
Immutable vs mutable

Mutable objects

```python
x = [1, 2, 3]
y = [1, 2, 3]

print(x == y)  # True
print(x is y)  # False
print(x[0] is y[0])  # True
```
immutable vs mutable

With simpler data types, **immutability** is useful. (no side effects)

With complex data types, **mutability** and **aliasing** is useful. (avoid copying large data)

Suppose you have a list of names.
You add another name to the list

\[\text{copy the whole list.} \]
If lists were immutable:

\[
x = ["Alice", "Bob", "Charlie", "David", \ldots \ldots]\n\]

a million names
If lists were immutable:

\[ x = \left[ \text{“Alice”, “Bob”, “Charlie”, “David”, ……} \right] \]

\[ x += \left[ \text{“Jordan”} \right] \]
But lists are **mutable**

\[
x = ["Alice", "Bob", "Charlie", "David", ........]
\]

\[
x += ["Jordan"]
\]
```python
def square(x):
    x = x**2
    return x

n = 5
squaredNum = square(n)
print(n, squaredNum)
    5   25

def square(a):
    for i in range(len(a)):
        a[i] = a[i]**2
    return a

b = [1, 2, 3]
squaredList = square(b)
print(b, squaredList)
    [1, 4, 9] [1, 4, 9]

Original b is destroyed
```
def square(x):
    x = x**2
    return x

n = 5
squaredNum = square(n)
print(n, squaredNum)
  5  25

import copy
def square(a):
    a = copy.copy(a)
    for i in range(len(a)):
        a[i] = a[i]**2
    return a

b = [1, 2, 3]
squaredList = square(b)
print(b, squaredList)
  [1, 2, 3] [1, 4, 9]

Original b is not destroyed
Strings vs Lists

names = "Alice,Bob,Charlie,…"  a million names

Suppose you want to change Bob to William.

```python
def changeName(s, oldName, newName):
    return s.replace(oldName, newName)
```

Create a new string with a million names.

changeName(names, "Bob", "William")

names and s are aliases.  
but this doesn’t have side-effects (names doesn’t change).
immutable vs mutable

**Strings vs Lists**

names = [“Alice”, “Bob”, .......]  

```python
def changeName(a, oldName, newName):
    for index in range(len(a)):
        if (a[index] == oldName):
            a[index] = newName

changeName(names, “Bob”, “William”)
```

*a million names*

names and *a* are aliases.

changes to *a* also change *names*.

The list of names is never duplicated/recreated.
Strings vs Lists

Immutable ----> make copy every time you change it.

If dealing with huge strings, or need to modify a string many times:

convert the string to a list first:

```python
longText = list("Once upon a time, in a land far far away...")
```

converting the list back to a string:

```python
longTextString = "".join(longText)
```
British mathematician, logician, cryptanalyst, computer scientist.

Father of computer science and artificial intelligence.
List operators and methods

2 types:

Destructive
- modifies original list

Non-destructive
- does not modify original list
- creates a new list
    (with strings, for example, this is what happens)
# List operators and methods

## Adding elements

<table>
<thead>
<tr>
<th>Destructive</th>
<th>NonDestructive</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a = [1, 2, 3])</td>
<td>(a = [1, 2, 3])</td>
</tr>
<tr>
<td>(a.append(4))</td>
<td>(b = a + [4])</td>
</tr>
<tr>
<td>(a = [1, 2, 3, 4])</td>
<td>(b = [1, 2, 3, 4])</td>
</tr>
<tr>
<td>(a.extend([5, 6]))</td>
<td>(c = b + [5, 6])</td>
</tr>
<tr>
<td>(a = [1, 2, 3, 4, 5, 6])</td>
<td>(c = [1, 2, 3, 4, 5, 6])</td>
</tr>
<tr>
<td>(a += [7, 8]) # same as extend</td>
<td>(b = [1, 2, 3, 4])</td>
</tr>
<tr>
<td>(a = [1, 2, 3, 4, 5, 6, 7, 8])</td>
<td>(c = [1, 2, 3, 4, 5, 6])</td>
</tr>
<tr>
<td>(a.insert(1, 1.5))</td>
<td>(d = c[:1] + [1.5] + c[1:])</td>
</tr>
<tr>
<td>(a = [1, 1.5, 2, 3, 4, 5, 6, 7, 8])</td>
<td>(d = [1, 1.5, 2, 3, 4, 5, 6])</td>
</tr>
</tbody>
</table>
IMPORTANT!

```
a = [1, 2, 3]
b = a
a += [4]
print(a)  # [1, 2, 3, 4]
print(b)  # [1, 2, 3, 4]
```

```
a = [1, 2, 3]
b = a
a = a + [4]
print(a)  # [1, 2, 3, 4]
print(b)  # [1, 2, 3]
```

```
a += [4]  # not same as

not same as
a = a + [4]
```
List operators and methods

Removing elements

<table>
<thead>
<tr>
<th>Destructive</th>
<th>NonDestructive</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = [1, 2, 3, 1, 2, 3, 1, 2, 3]</td>
<td>a = [2, 1, 2, 1, 2]</td>
</tr>
<tr>
<td>a.remove(3)</td>
<td>b = a[:1] + a[3:]</td>
</tr>
<tr>
<td>a = [1, 2, 1, 2, 3, 1, 2, 3]</td>
<td>b = [2, 1, 2]</td>
</tr>
<tr>
<td>a.remove(3)</td>
<td>a = [2, 1, 2, 1, 2]</td>
</tr>
<tr>
<td>a = [1, 2, 1, 2, 1, 2, 3]</td>
<td></td>
</tr>
<tr>
<td>a.pop()</td>
<td></td>
</tr>
<tr>
<td>a = [1, 2, 1, 2, 1, 2]</td>
<td></td>
</tr>
<tr>
<td>print(a.pop(0))</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a = [2, 1, 2, 1, 2]</td>
<td></td>
</tr>
<tr>
<td>a[1:3] = [ ]</td>
<td></td>
</tr>
<tr>
<td>a = [2, 1, 2]</td>
<td></td>
</tr>
<tr>
<td>a = [2, 1:3]</td>
<td></td>
</tr>
<tr>
<td>del a[1:]</td>
<td></td>
</tr>
<tr>
<td>a = [2]</td>
<td></td>
</tr>
</tbody>
</table>
def remove(someList, element):
    for index in range(len(someList)):
        if (someList[index] == element):
            someList.pop(index)

def total(someList):
    t = 0
    while(someList != []):
        t += someList.pop()
    return t

da = [1, 2, 3, 1, 2, 3, 1, 2, 3]
print(total(a))   18
print(a)         []

Never change the list if you don’t need to!

Index range changes every time you pop.
List operators and methods

**sort vs sorted**

<table>
<thead>
<tr>
<th>Destructive</th>
<th>NonDestructive</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = [1, 2, 3, 1, 2, 3]</td>
<td>a = [1, 2, 3, 1, 2, 3]</td>
</tr>
<tr>
<td>a.sort()</td>
<td>b = sorted(a)</td>
</tr>
<tr>
<td>a = [1, 1, 2, 2, 3, 3]</td>
<td>b = [1, 1, 2, 2, 3, 3]</td>
</tr>
<tr>
<td></td>
<td>a = [1, 2, 3, 1, 2, 3]</td>
</tr>
</tbody>
</table>
List operators and methods

finding an element

```
a = [1, 2, 3, 1, 2, 3]

print(a.index(2))  # 1
print(a.find(2))   # ERROR: no method called 'find'
print(a.index(4))  # ERROR: 4 is not in the list

if (4 in a):
    print("4 is at index", a.index(4))
else:
    print("4 is not in the list.")
```
List operators and methods

others

https://docs.python.org/3/library/stdtypes.html#typesseq-mutable

https://docs.python.org/3/tutorial/datastructures.html#more-on-lists
Summary

**Destructive** (modifies the given list)

+,
+=

every method that manipulates the list

del statement

**NonDestructive**

+, *

functions

slicing

Be careful about aliasing (especially with function parameters)
Tuples

The immutable brother of lists
Tuples

myTuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

myTuple = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10  # not recommended

myTuple = (1, "hello", 3.14, True)

myTuple = (1,)   # Put comma for one element tuple

myTuple[0] = 2   ERROR

parallel assignments
(x, y) = (1, 2)
Tuples

return multiple values in a function

```python
def firstPrimeInList(a):
    for i in range(len(a)):
        if isPrime(a[i]):
            return (i, a[i])
    return -1
```
Exercise Problem
Lockers Problem

1 2 3 4 5 6 7

...

n

...

...

...