15-112
Fundamentals of Programming

Week 2 - Lecture 3:
Lists
## Builtin 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>s = “hw2-1 was hard”</td>
<td>a = [1, 3.14, “hi”, True]</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>s[0] = “H”</td>
<td>a[0] = 100</td>
</tr>
</tbody>
</table>
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))  # 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
print(e[::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

```
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

Memory (Storage)
Closer look at RAM (Main memory)

Address

memory cell

2843
2844
2845
2846
2847
2848
2849
2850
Immutable vs mutable

Immutable objects

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

Immutable objects

\[
x = 5
\]
\[
y = 4
\]
\[
x = 1
\]
\[
y -= 2
\]

\[
x \rightarrow 2843
\]
\[
2844
\]
\[
2845
\]
\[
2846
\]
\[
y \rightarrow 2847
\]
\[
2848
\]
\[
2849
\]
\[
2850
\]
\[
\vdots
\]
immutable vs mutable

Immutable objects

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

\[
\begin{array}{c|c}
\hline
x & 2843 \\
2844 & 1 \\
2845 & 5 \\
2846 & \ \\
2847 & 4 \\
2848 & \ \\
2849 & \ \\
\hline
y & 2850 \\
\hline
\end{array}
\]
### Immutable Objects

<table>
<thead>
<tr>
<th>x</th>
<th>2843</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>2844</td>
</tr>
<tr>
<td></td>
<td>2845</td>
</tr>
<tr>
<td></td>
<td>2846</td>
</tr>
<tr>
<td></td>
<td>2847</td>
</tr>
<tr>
<td></td>
<td>2848</td>
</tr>
<tr>
<td></td>
<td>2849</td>
</tr>
</tbody>
</table>

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

- Garbage

---

```python
x = 5
y = 4
x = 1
y -= 2
```
immutable vs mutable

Immutable objects

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

“Garbage is collected”
immutable vs mutable

Immutable objects

\[ x = 5 \quad \rightarrow \quad x \rightarrow 5 \]

\[ y = 4 \quad \rightarrow \quad y \rightarrow 4 \]
Immutable vs mutable

Immutable objects

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

\[
\begin{align*}
  &\quad \rightarrow \quad 5 \\
  &\quad \rightarrow \quad 4
\end{align*}
\]
Immutable vs mutable

Immutable objects

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

Immutable objects

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

Immutable objects

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

Immutable objects

\[ x = 5 \]

\[ x \rightarrow 5 \]
immutable vs mutable

Immutable objects

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

Immutable objects

```python
x = 5
y = x
x += 1
print(x, y)
```

- `x = 5` creates an immutable object.
- `y = x` assigns a reference to `x`, and `x` is also mutable.
- `x += 1` changes the value of `x`, but `y` still points to the original value of `x`.

This illustrates the difference between immutable and mutable objects in Python.
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:

\[ x \rightarrow 5 \]
\[ y \rightarrow 5 \]

In practice:

\[ x \rightarrow 5 \]
\[ y \rightarrow 5 \]

(seems like a good thing)
immutable vs mutable

Mutable objects

$x = [1, 2, 3]$

So actually, a list is a sequence of references (variables)!
immutable vs mutable

Mutable objects

\[\begin{align*}
x &= [1, 2, 3] \\
y &= x
\end{align*}\]
**immutable vs mutable**

**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
print(y[0])  # 4

x and y are **aliases**.
immutable vs mutable

Mutable objects

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

print(x == y)   True
print(x is y)   False
print(x[0] is y[0])   True
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

Don't copy the whole list.
**immutable vs mutable**

**If lists were **immutable**:**

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

---

**Diagram:**

```
x ['Alice', 'Bob', 'Charlie', 'David', \ldots \ldots]
```

---

**Text:**

A million names
If lists were immutable:

\[ x = [\text{“Alice”}, \text{“Bob”}, \text{“Charlie”}, \text{“David”}, \ldots] \]

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

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

\[
x += [\text{“Jordan”}]
\]
def square(x):
    x = x**2
    return x

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

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
immutable vs mutable

Strings vs Lists

names = “Alice,Bob,Charlie,…”  a million names

Suppose you want to change Bob to William:

names = names.replace(“Bob”, “William”)

Creates a new string with a million names.
**immutable vs mutable**

**Strings vs Lists**

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

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

```python
names = [“Alice”, “Bob”, .......]
```

*a million names*

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

```
names and a are aliases.
changes to a affect names.
```

The list of names is never duplicated/recreated.
Immutable vs Mutable

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)
  ```
Digression: Alan Turing (1912-1954)

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

Destructive

\[
a = [1, 2, 3]
\]
\[
a.append(4)
\]
\[
a = [1, 2, 3, 4]
\]
\[
a.extend([5, 6])
\]
\[
a = [1, 2, 3, 4, 5, 6]
\]
\[
a += [7, 8] \quad \# \text{same as extend}
\]
\[
a = [1, 2, 3, 4, 5, 6, 7, 8]
\]
\[
a.insert(1, 1.5)
\]
\[
a = [1, 1.5, 2, 3, 4, 5, 6, 7, 8]
\]

NonDestructive

\[
a = [1, 2, 3]
\]
\[
b = a + [4]
\]
\[
b = [1, 2, 3, 4] \quad a = [1, 2, 3]
\]
\[
c = b + [5, 6]
\]
\[
c = [1, 2, 3, 4, 5, 6] \quad b = [1, 2, 3, 4]
\]
\[
d = c[:1] + [1.5] + c[1:]
\]
\[
d = [1, 1.5, 2, 3, 4, 5, 6] \quad c = [1, 2, 3, 4, 5, 6]
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]

**IMPORTANT!**

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

Removing elements

**Destructive**

a = [1, 2, 3, 1, 2, 3, 1, 2, 3]
a.remove(3)
   a = [1, 2, 1, 2, 3, 1, 2, 3]
a.remove(3)
   a = [1, 2, 1, 2, 1, 2, 3]
a.pop()
   a = [1, 2, 1, 2, 1, 2]
print(a.pop(0))
   1
   a = [2, 1, 2, 1, 2]
a[1:3] = []
   a = [2, 1, 2]
del a[1:]
   a = [2]

**NonDestructive**

a = [2, 1, 2, 1, 2, 1, 2]
b = a[:1] + a[3:]
   b = [2, 1, 2]    a = [2, 1, 2, 1, 2]
List operators and methods

Common Mistakes

def remove(someList, element):
    for index in range(len(someList)):
        if (someList[index] == element):
            someList.pop(index)

    never change the list
    if you don't need to!

Common Mistakes

    Index range changes
every time you pop.

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

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

    a = [1, 2, 3, 1, 2, 3, 1, 2, 3]
print(total(a))
print(a)  []
List operators and methods

**sort vs sorted**

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

…