Scope, Aliasing, Tuples & Mutability

Intro2CS – week 4a
Scope (quick reminder)
Scope

• Variables inside a function are local. They are not recognized or known outside of it.
  – Hides internal structure
  – Makes function more independent of code outside

```python
def a(x):
    y = x + 2
    return y
```

```python
a(1)
print(x)  # Error!
print(y)  # Error!
```
Scope

On the other hand, a function knows of variables that are outside.

\[ y = 2 \]
\[ \text{def } a(): \]
\[ \quad \text{print}(y) \]
\[ y = 3 \]
\[ a() \quad \text{Will work, and print the value of } y \text{ (that is 3)} \]

Very confusing. Try to avoid it (except with constants). Instead, give needed values as parameters!
Scope

• But, functions do not usually change variables outside. This can be very bad for code readability:

\[ a=1 \]
\[ b=3 \]
\[ c=4 \]

\[ c = func(b) \]

Did any of my variables change? Which ones? We ASSUME that only \( c \) changes
Scope

- Python assumes that if you are assigning a value to a variable inside a function, you mean a new local variable (that happens to have the same name):

  ```python
  x=1
  y=2
  def f(x):
      x=x+1
      y=x+1
  f(1)
  print(x)
  print(y)
  ```

  **External scope:**
  ```text
  x,y
  ```

  **Function a(x):**
  ```text
  x,y
  ```

  External variables unchanged. (Prints 1,2)
Side effects and global variables

• If you insist, you can indeed reach out and touch an external “global” variable:

```python
x=1
def a():
    global x
    x=x+1
a()
a()
print(x)
```

This is very very bad code! Avoid this!

Will print 3
Scope

- Scope can be more complex, and have several nested levels:

```python
def f1(x):
    def f2(y):
        return y**2
    return f2(f2(x))

def f3(x):
    print(x)
```

External scope:
- Function f1(x):
  - Function f2(y):
  - Function f3(x):
References, Aliases and Copies
References

• It is important to understand how information is represented internally.

• Variables do not directly hold a value, but instead hold its address in memory.
  – a “reference” to the value / object.

\[
x = 3 \\
y = "hello"
\]
How we often think about this:
Variables **point** to the value.

\[
\begin{align*}
x &= \text{"hello"} \\
y &= 3
\end{align*}
\]

(We usually don’t care about the address in memory)
Assignment copies **the address** from one variable to the other.

```python
x = "hello"
y = x
print(id(x))
Print(id(y))
```

**Builtin function id will show you the number of the memory cell (in most python implementations)**

```
x: 22130
"Hello"
y: 22130
```
Lists

Lists (and other containers) also work the same way. They reference the values / objects inside them.

\[ x = [1,"hello",2.0] \]
Lists

Implications:

```
x = [1,"hello",2.0]
y = x

y[0] = 53
print(x[0])
#output will be 53
```
Copying the list

In contrast to the previous examples...

```python
x = [1, "hello", 2.0]
y = x[:]
y[0] = 53
print(x[0])
# output will be 1
```

Slicing creates a brand new list!

A “shallow copy”
Comparing two lists

Two questions we might ask:
1. Are the two lists actually the same list? (the `is` operator)
2. Are the contents of the two lists the same? (the `==` operator)

\[
x = [1, \text{"hello"}, 2.0]
y = x[:]
z = x
\]

print(\(x \text{ is } y\))  # False
print(\(x == y\))  # True
print(\(z \text{ is } x\))  # True
append() vs. +

• The + operator on lists creates a new list

```python
x = [1,2,3]
y = x + [4,5,6]

print(x is y)  #False
```

• append() and extend() do not

```python
x = [1,2,3]
x.extend([4,5,6])
x.append(7)
```
How else can we copy lists?

```python
x = [1, 2, 3]
y = copy.copy(x)
z = list(x)

print(y is x)  # False
print(z is x)  # False
```
Tuples

- Tuples are another sequence datatype
- Just like lists
- Except **they are immutable**. cannot be changed.

```python
x = (0, 1, 2)
x = 0, 1, 2  # equivalent
u = (3,)
    # a tuple of size 1

print(len(x))  # length
y = tuple(range(3))  # constructor
z = x + y  # z is a new tuple
print(z[2:4])  # slicing works

for item in x: print(item)  # works too

x[0] = 3  # ERROR!
x.append(2)  # ERROR!
```
Unpacking

coord = (1, 2, 3)
x, y, z = coord

print(x)

Allows assignments to multiple variables

So actually...
functions have a single return value

```python
def func():
    return (1,2)

x, y = func()
```

This is tuple unpacking.
Functions can accept lists as parameters:

```python
def set_zero(lst, i):
    lst[i] = 0

my_list = [1, 2, 3]
set_zero(my_list, 1)
print(my_list)
```

**Result:** Function alters list, in outer scope, but *without having to return it*.

Must be aware of this when we program!
Mutable and Immutable Types

• We can pass immutable types to a function knowing that they will not be changed.

```python
def funky_function(tpl):
    ... some code ...

my_tpl = 1, 2, 3
funky_function(my_tpl)
print(my_tpl)
```
Strings are immutable too.

```python
x = "hi"
y = "bye"

z = x + y  # z is a new string!

w = z[2:4]  # slicing doesn’t change original strings...

x[0] = 3    # ERROR!
x.append("a")  # ERROR!
```

So are ints, floats, booleans, ranges
Immutable types are often reused
(and sometimes not)

```
In[54]: x = "hello"
In[55]: y = "he" + "llo"
In[56]: x is y
Out[56]: True
In[57]: x = (0,1,2)
In[58]: y = (0,) + (1,2)
In[59]: x is y
Out[59]: False
In[60]: x==y
Out[60]: True
```
How to deep copy lists

```python
original = [['Moe', 100], ['Joe', 30]]
alias = original

original[1][1] = "CHANGED"
original.append(['Fred', 95])
print(alias)
```

```
"Moe" 100  "Joe" 30  "Fred" 95
"CHANGED"
```

```
original = [['Moe', 100], ['Joe', 30]]
alias = original

original[1][1] = "CHANGED"
original.append(['Fred', 95])
print(alias)
```
How to deep copy lists

```
original = [['Moe', 100], ['Joe', 30]]
alias = original

shallow_cpy = original[:]

original[1][1] = "CHANGED"
original.append(['Fred', 95])

print(alias) ; print(shallow_cpy)
```
How to deep copy lists

```python
original = [['Moe', 100], ['Joe', 30]]
alias = original
shallow_cpy = original[:]
depth_cpy = copy.deepcopy(original)

original[1][1] = "CHANGED"
original.append(['Fred', 95])

print(alias) ; print(shallow_cpy) ; print(deep_cpy)
```
How to deep compare

== does deep comparison.
(It uses == to compare all cells of a list)

```python
original = [['Moe', 100], ['Joe', 30]]
depth_cpy = copy.deepcopy(original)

print(depth_cpy is original)  # False
print(depth_cpy[0] is original[0])  # False
print(depth_cpy == original)   # True
```
Mind benders

a is a tuple. It is immutable.

```
a = ([1, 2, 3], "hi", 2.0)
a[0][0] = 52
```

is this going to run?

Did we change the contents of the tuple?
Mind benders

• What does the following code do?

```python
a = [1, 2]
b = [1, 2]
a[0] = b
b[0] = a
print(a[0])
```

• Can a list contain itself?
Programs with multiple files
Importing files

Suppose `foo.py` has code we want to use in `bar.py`

In `bar.py` we can write:

```python
import foo
foo.func1()
```

```python
import foo as my_foo
my_foo.func1()
```

```python
from foo import func1, func2
func1()
```

```python
from foo import *
func1()
```

Here’s more info on handling cyclic imports. https://docs.python.org/3/faq/programming.html#how-can-i-have-modules-that-mutually-import-each-other
Importing runs the code

We want to be able to run foo.py, but also want to use its code in bar.py

```
import foo

foo.func1()
...more_stuff...
```

```
def func1():
    dostuff...
    print("hello")
```

If we run this, also prints.

Run this. Okay.

```
if __name__ == "__main__":
    print("hello")
```

__name__ will have the value "__main__" only if the current module is first to execute.

Run this. Okay.
Function Extras
Default values for arguments

def p_print(text, times=1, p_char="*" ):
    print(p_char * (len(text)+4))
    for i in range(times):
        print(p_char, text, p_char)
    print(p_char * (len(text)+4))

p_print("hello")
p_print("hello",2)
p_print("hello",2,'@')
p_print("hello",p_char = '@')
p_print(p_char = '@', text = "hello", times=2)
p_print(p_char = '@','hello')  #ERROR!
Unknown number of arguments

• You can define functions that accept as many arguments as the user wishes to provide.
• Arguments are placed in a tuple.

```python
def is_word_in(word, *text_list):
    for text in text_list:
        if word in text:
            print(word, "is in", text)

is_word_in("run", "rune", "cat", "prune", "apple", "runner")
is_word_in("run")
```