FIT1045: Algorithms and Programming Fundamentals in Python

Lecture 7

Understanding Python

Announcements

Test 1 this week

- covers only material from Weeks 1 – 3
- questions similar to the exam
  - similar to workshop and tutorial tasks
- opens August 27 2am
- closes August 28, 1pm
- timed (45m)

Objectives

To understand

- variable and value representation in Python
- multiple assignment and tuples
- mutable versus immutable objects
- code execution in Python

This covers learning outcomes

- 3 – Analyse the behaviour of programs and data structures

What is Python?

https://www.thinglink.com/scene/619900611055321088

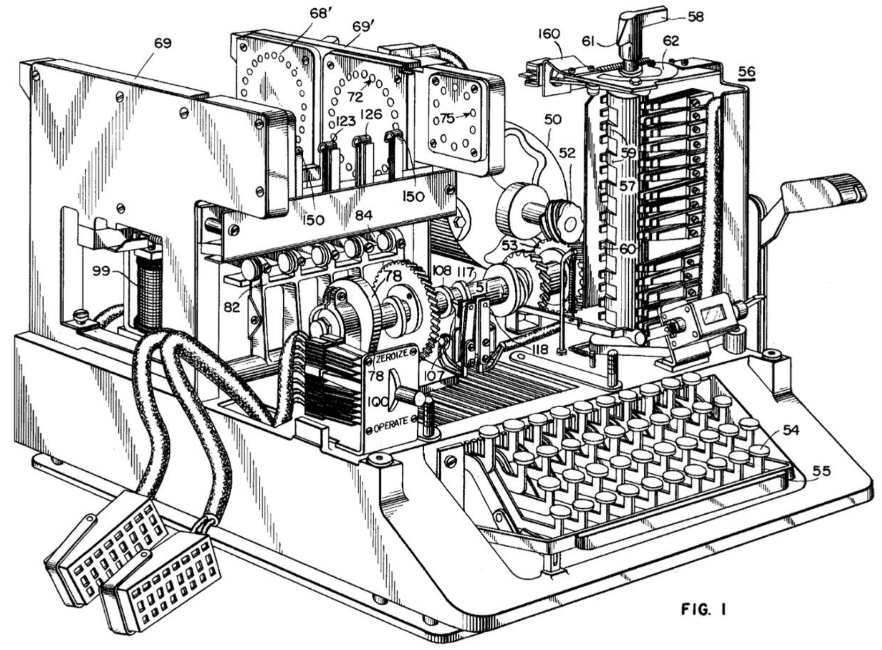
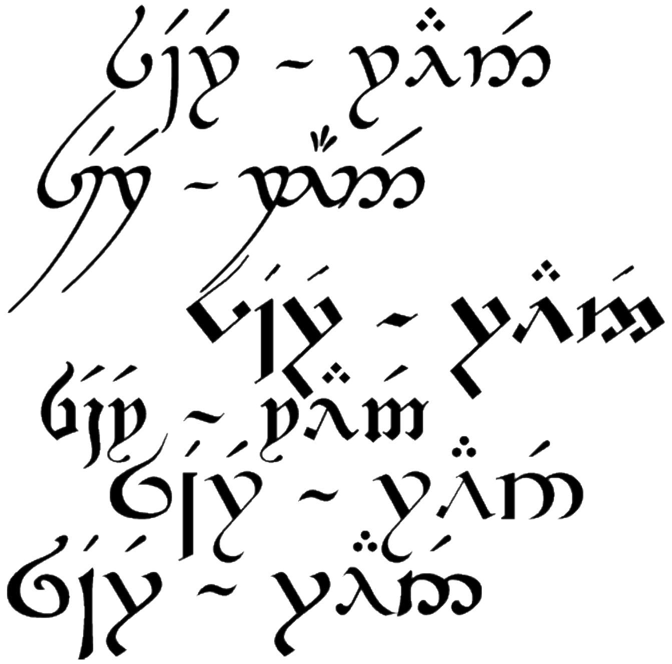


FIG. 1

A (virtual) Machine –
The Python Interpreter

https://www.pinterest.com.au/pin/269090146459263825/



A Language –
The Python programming language

Today: having a closer look here

Overview

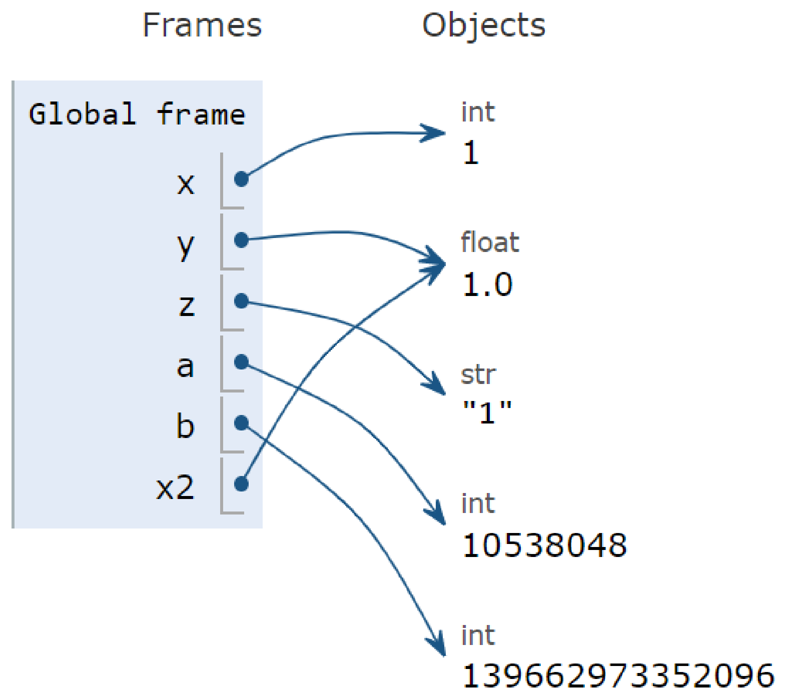
1. Objects and variables: things and names for things
2. Functions, multiple assignments, and tuples
3. Mutability: the same and the similar

What are Python *objects* and *variables*?

```
x = 1
y = 1.0
z = '1'

a = id(1)
b = id(1.0)

x2 = 1
x2 = 1.0
```



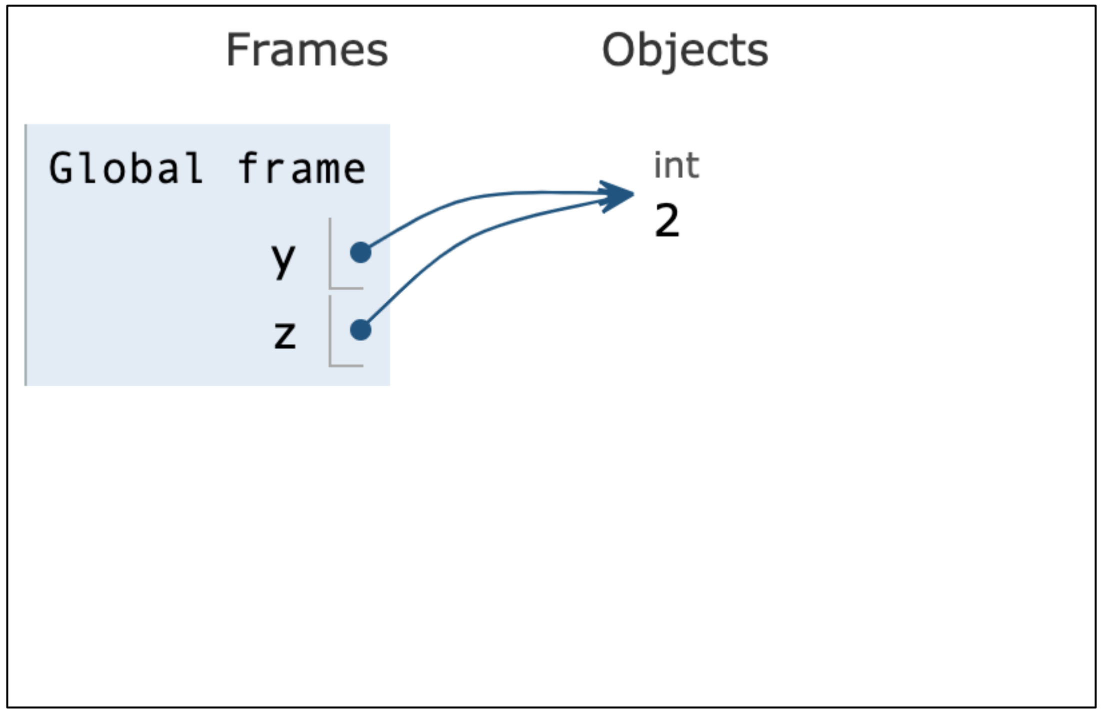
Python Tutor: https://goo.gl/Cg3hE9

- *objects* have: type, identity, value (type-specific content)
- *variable* is a name (identifier) and a reference to an object
- assignment operation creates variable *if necessary*
- object *potentially* created when evaluating expression (re-using certain immutable objects)
- variables can be re-assigned to another object (possibly of different type: variables don't have types themselves)

More on variables and objects

```
x = 1
y = x + 1
z = x

del x
z = y
```



Python Tutor: https://goo.gl/e8HCr1

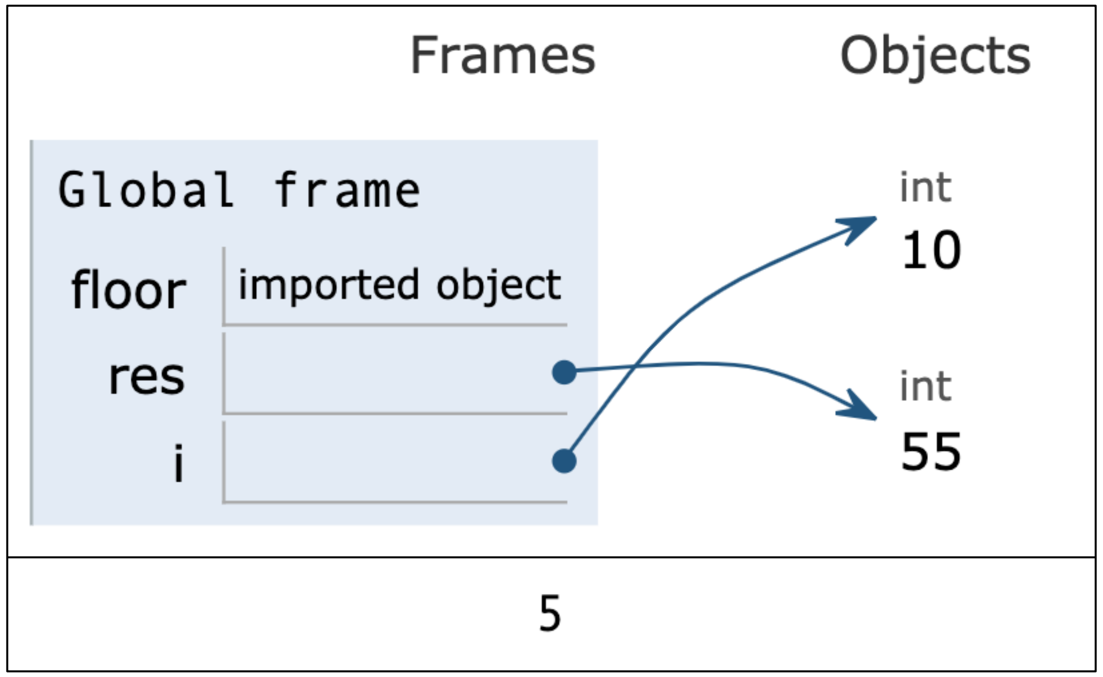
- variables are expressions that evaluate to referenced object
- assignment operator evaluates right-hand-side expression
- so when assigning a variable to another, assignment operator assigns to referenced object (no reference chain)
- variables (names) can be deleted by del-statement
- objects disappear (are deleted) when no longer referenced

What defines names?

```
from math import floor

res = 0
for i in range(1, 11):
    res = res + i

print(floor(res / i))
```



```
5
```

Python Tutor: https://goo.gl/Q3wWAk

- assignment operation
- import and for statement
- name known until explicitly deleted or frame discarded (when leaving function execution)
- another source: function definitions (see below)

Exercise: swapping references

```
x = 1
y = 2

x = y
y = x

print(x)
print(y)
```

https://flux.qa

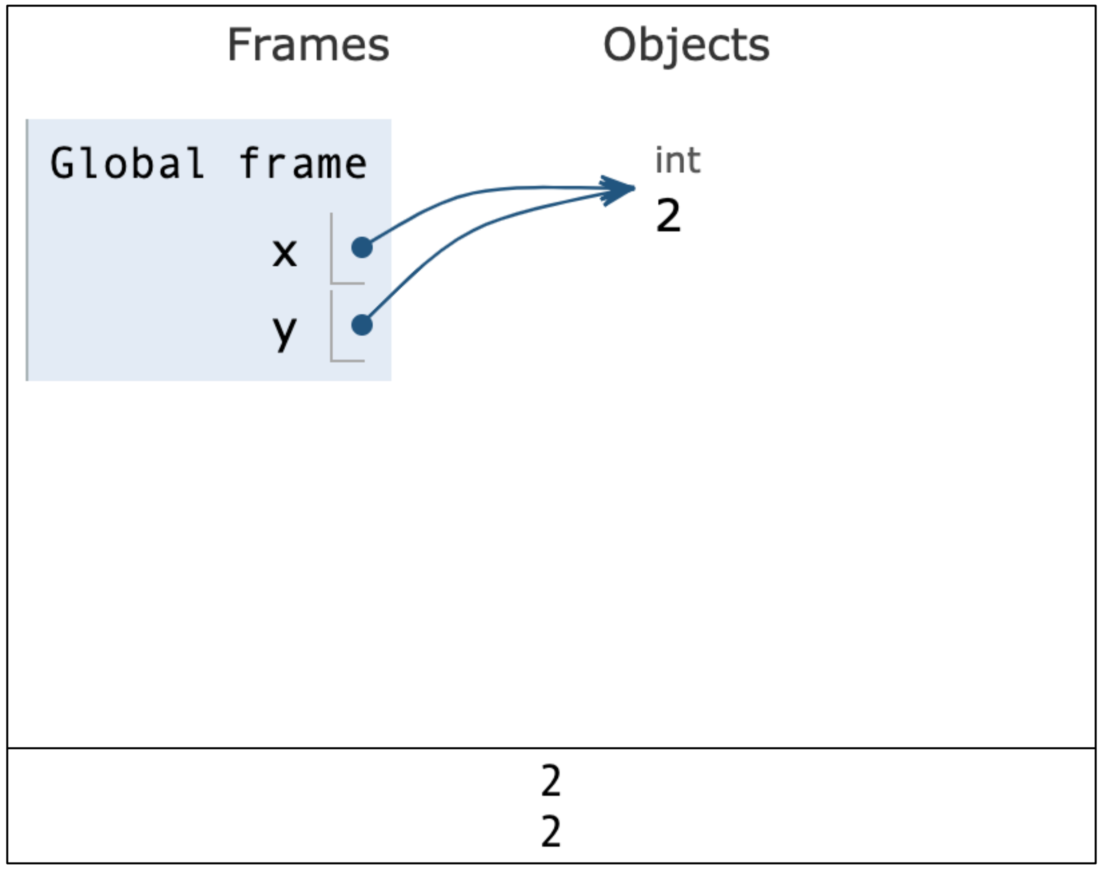
Clayton: AXXULH
Malaysia: LWERDE

Exercise: swapping references

```
x = 1
y = 2

x = y
y = x

print(x)
print(y)
```



```
2
2
```

Python Tutor: https://goo.gl/ihv8Px

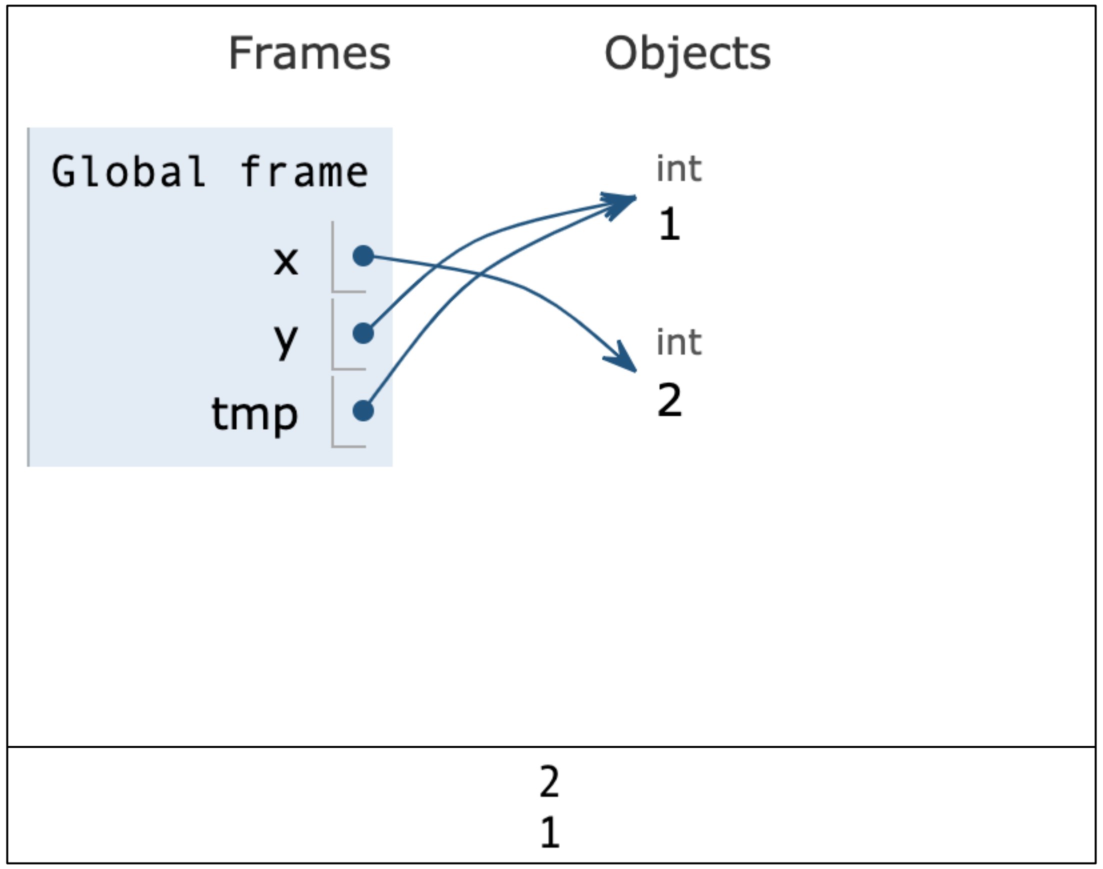
- once we lose all references to it, object is lost

Exercise: swapping references

```
x = 1
y = 2

tmp = x
x = y
y = tmp

print(x)
print(y)
```



```
2
1
```

Python Tutor: https://goo.gl/U4mvcP

- once we lose all references to it, object is lost
- need to keep track of it by storing temporary reference
- actually we already know more convenient way to swap (next section)

Overview

1. Objects and variables: things and names for things
2. Functions, multiple assignments, and tuples
3. Mutability: the same and the similar

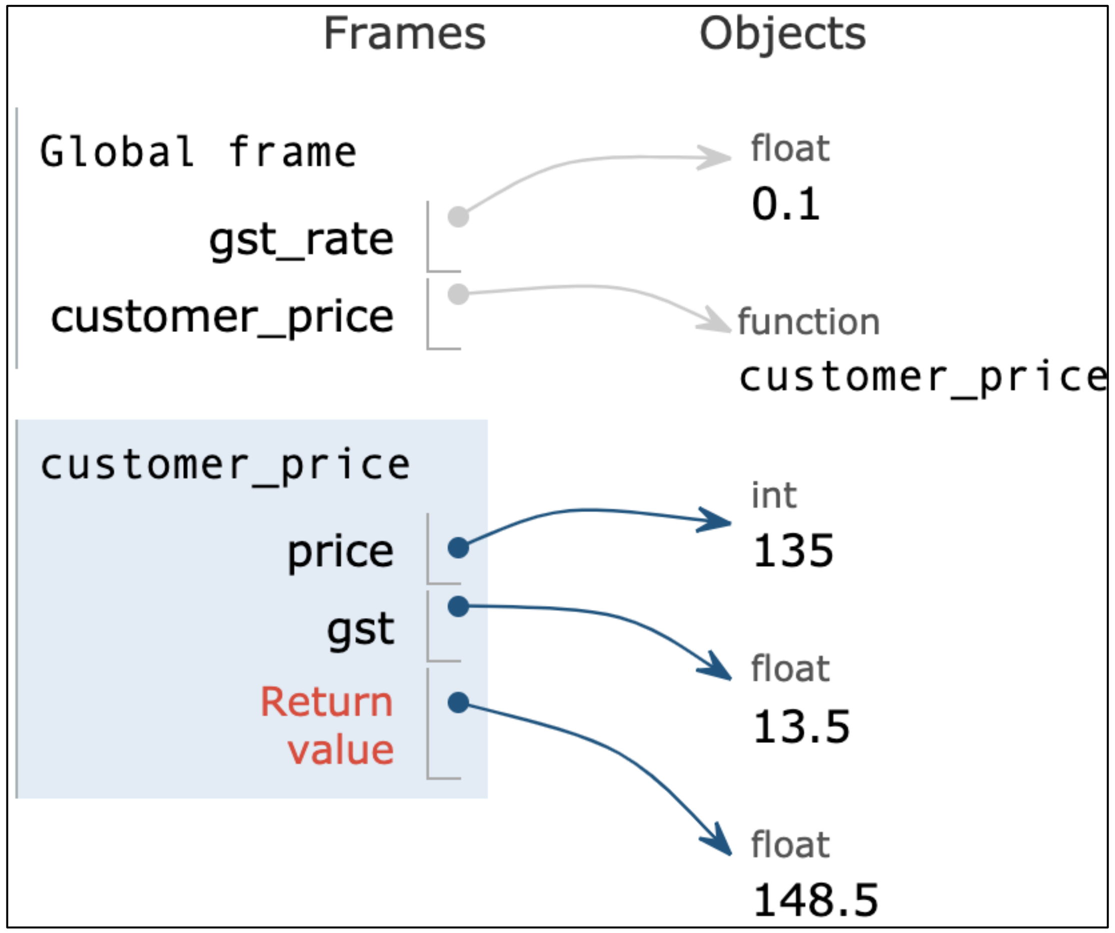
What are Python *functions*?

```
gst_rate = 0.1

def customer_price(price):
    gst = price * gst_rate
    return price + gst

p1 = customer_price(135)
```

Python Tutor: https://goo.gl/Yeu9ZV



- *functions* are also objects with an identifier
- def-statement creates (*or re-assigns*) matching variable
- function execution has own frame of local variables
- local variables for parameters are created on call

What are Python *functions*?

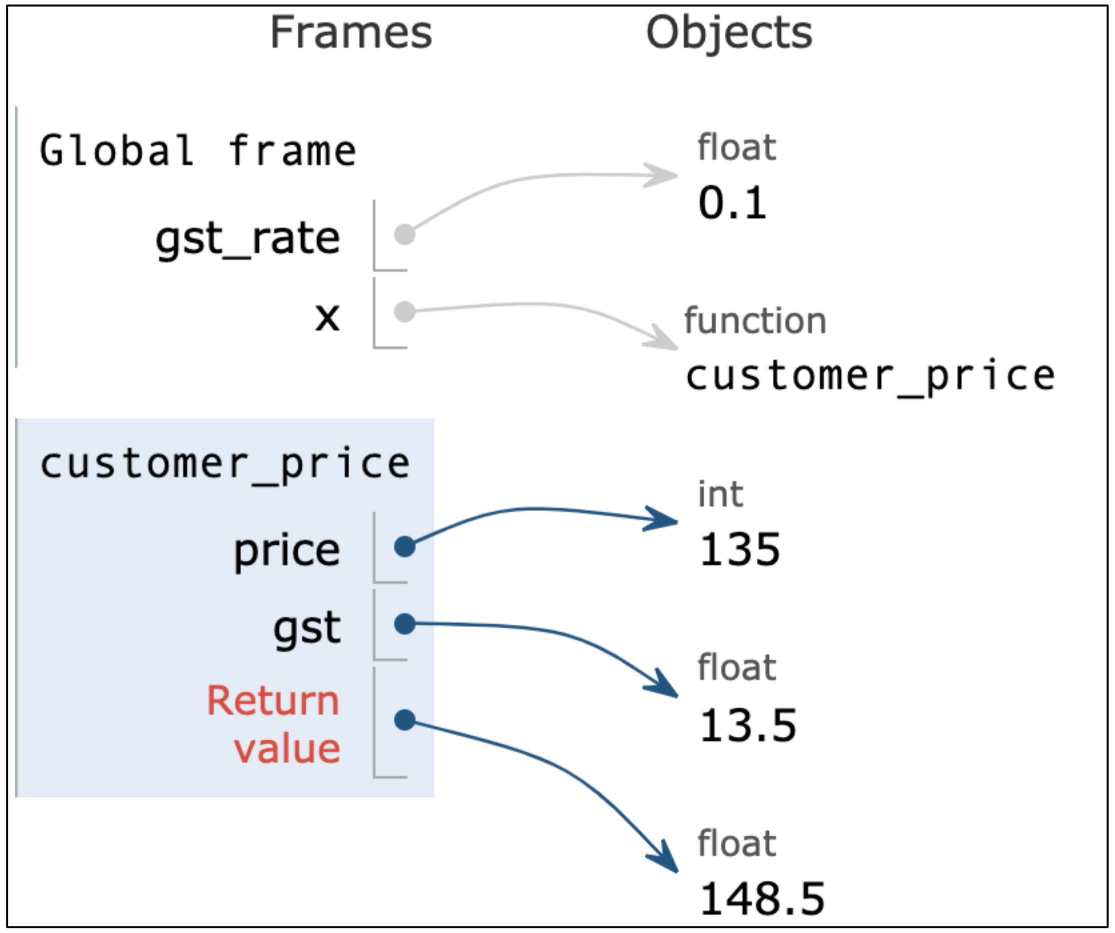
```
gst_rate = 0.1

def customer_price(price):
    gst = price * gst_rate
    return price + gst

x = customer_price
del customer_price

p1 = x(135)
```

Python Tutor: https://goo.gl/PaA3x9



- function objects can be assigned to other variables
- variables referring to function can be deleted (including the one created by def statement)
- can call function object under different names via the familiar call syntax

Functions are things, too

```
>>> round(1.5)
2
>>> round
<built-in function round>
>>> x = round
>>> x(1.5)
2
```

Functions are "things that can be touched", not only operations

So functions can also be parameters of other functions!

Recall: reading with type conversion

foods.txt

```
beef
potato
broccoli
apple
potato
apple
tofu
tomato
```

quantities.txt

```
300
300
200
100
250
100
120
200
```

```python
def list_from_file(fname, num=False):
    file = open(fname)
    rs = []
    for l in file:
        if num:
            rs = rs+[float(l.strip())]
        else:
            rs = rs+[l.strip()]
    file.close()
    return rs
```

```
>>> foods = list_from_file('foods.txt')
>>> foods
['apple', 'broccoli', 'beef', 'lamb', 'bread', 'potato', 'tofu',
'tomato']
>>> quantities = list_from_file('quantities.txt', True)
>>> quantities
[300.0, 300.0, 200.0, 100.0, 250.0, 100.0, 120.0, 200.0]
>>>
```

Could add more cases in conditional

foods.txt

```
beef
potato
broccoli
apple
potato
apple
tofu
tomato
```

quantities.txt

```
300
300
200
100
250
100
120
200
```

```python
def list_from_file(fname, typ='str'):
    file = open(fname)
    rs = []
    for l in file:
        if typ=='float':
            rs = rs+[float(l.strip())]
        elif typ=='int':
            rs = rs+[int(l.strip())]
        else:
            rs = rs+[l.strip()]
    file.close()
    return rs
```

```
>>> foods = list_from_file('foods.txt')
>>> foods
['apple', 'broccoli', 'beef', 'lamb', 'bread', 'potato', 'tofu',
'tomato’]
>>> quantities = list_from_file('quantities.txt', 'float')
>>> quantities
[300.0, 300.0, 200.0, 100.0, 250.0, 100.0, 120.0, 200.0]
>>> quantities = list_from_file('quantities.txt', 'int')
>>> quantities
[300, 300, 200, 100, 250, 100, 120, 200]
```

Better solution: function parameter

foods.txt

```
beef
potato
broccoli
apple
potato
apple
tofu
tomato
```

quantities.txt

```
300
300
200
100
250
100
120
200
```

```
def list_from_file(filename, typ=str):
    file = open(filename)
    res = []
    for line in file:
        res = res + [typ(line.strip())]
    file.close()
    return res
```

higher-order function

function argument

```
>>> foods = list_from_file('foods.txt')
>>> foods
['apple', 'broccoli', 'beef', 'lamb', 'bread', 'potato', 'tofu',
'tomato’]
>>> quantities = list_from_file('quantities.txt', float)
>>> quantities
[300.0, 300.0, 200.0, 100.0, 250.0, 100.0, 120.0, 200.0]
>>> quantities = list_from_file('quantities.txt', int)
>>> quantities
[300, 300, 200, 100, 250, 100, 120, 200]
```

Another example for higher order function: built-in function *map*

```
>>> map(round, [1.1, 2.7, 2.3])
<map object at 0x10da21290>
>>> m = map(round, [1.1, 2.7, 2.3])
>>> type(m)
<class 'map'>
>>> for x in m: x
1
3
2
```

function as argument

converts one Iterable to another by applying function to each element

Another example for higher order function: built-in function *map*

```
>>> map(round, [1.1, 2.7, 2.3])
<map object at 0x10da21290>
>>> m = map(round, [1.1, 2.7, 2.3])
>>> type(m)
<class 'map'>
>>> for x in m: x
1
3
2
```

Provides yet another solution for type conversion

```
>>> foods = list_from_file('foods.txt')
>>> foods
['apple', 'broccoli', 'beef', 'lamb', 'bread', 'potato', 'tofu',
'tomato’]
>>> quantities = list_from_file('quantities.txt')
>>> quantities
['300', '300', '200', '100', '250', '100', '120', '200']
>>> list(map(int, quantities))
[300, 300, 200, 100, 250, 100, 120, 200]
```

Exercise 2: swapping via function

```
x = 1
y = 2

def swap(x, y):
    tmp = x
    x = y
    y = tmp

swap(x, y)
print(x)
print(y)
```

https://flux.qa

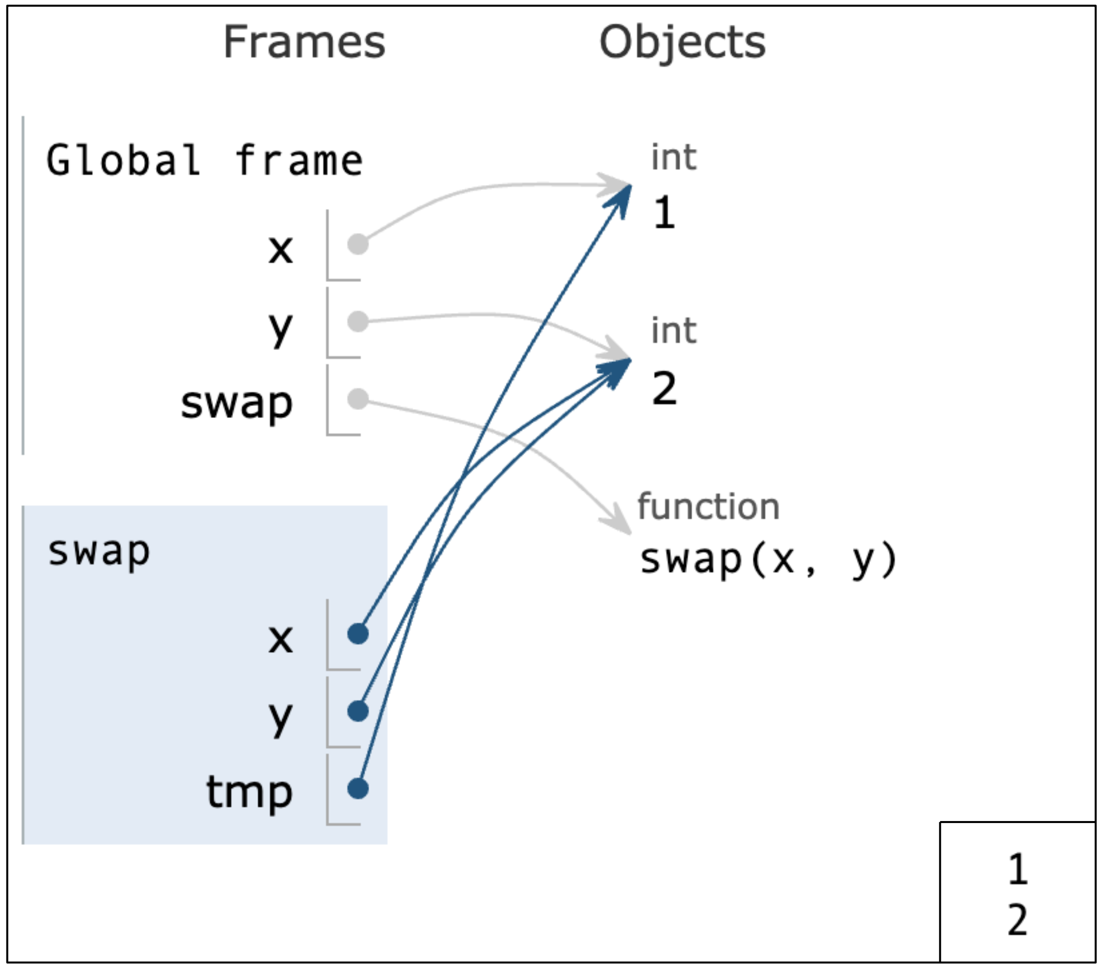
Clayton: AXXULH
Malaysia: LWERDE

Exercise 2: swapping via function

```
x = 1
y = 2

def swap(x, y):
    tmp = x
    x = y
    y = tmp

swap(x, y)
print(x)
print(y)
```



```
1
2
```

Python Tutor: https://goo.gl/rWij55

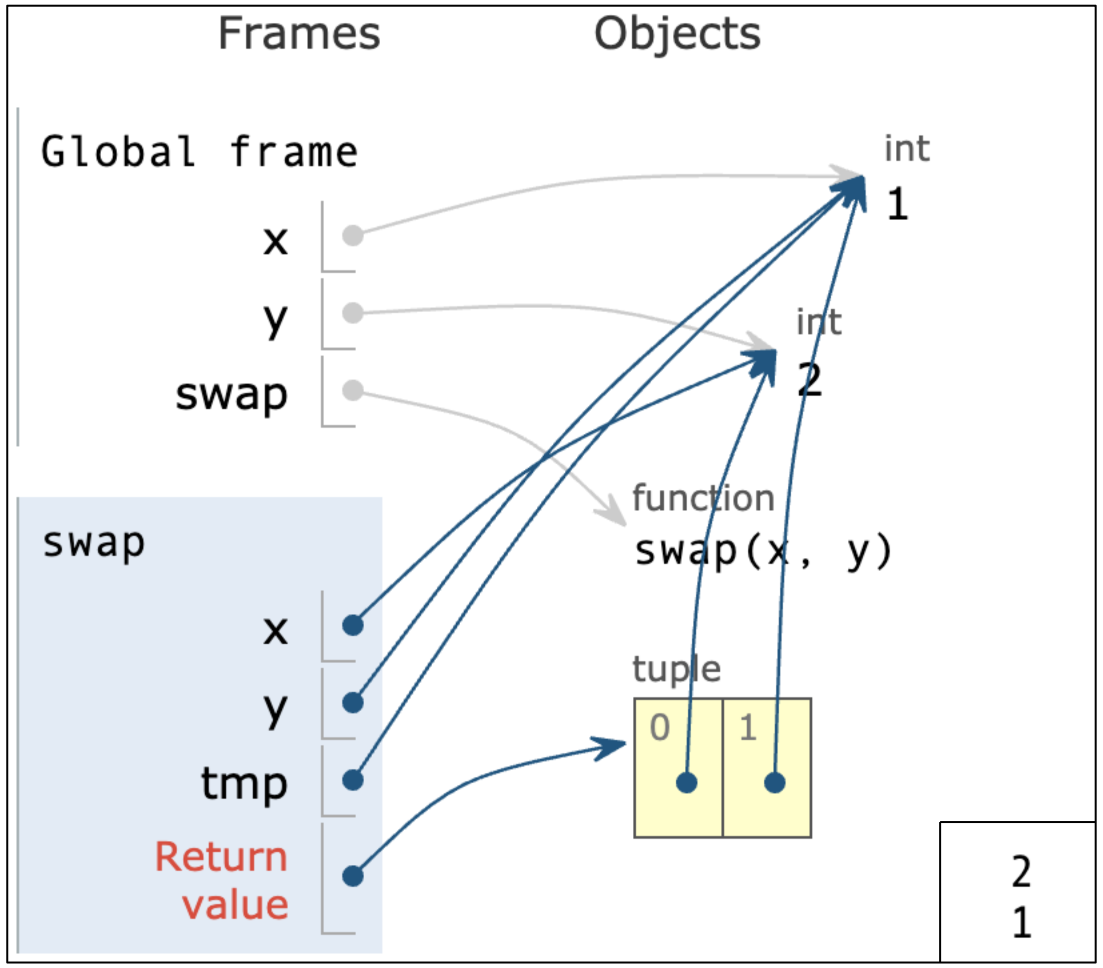
- local names shadow (hide) global names
- this function only swaps local variables
- functions can read global variables (but not directly re-assign them)

Exercise 2: swapping via function

```
x = 1
y = 2

def swap(x, y):
    tmp = x
    x = y
    y = tmp
    return x, y

x, y = swap(x, y)
print(x)
print(y)
```

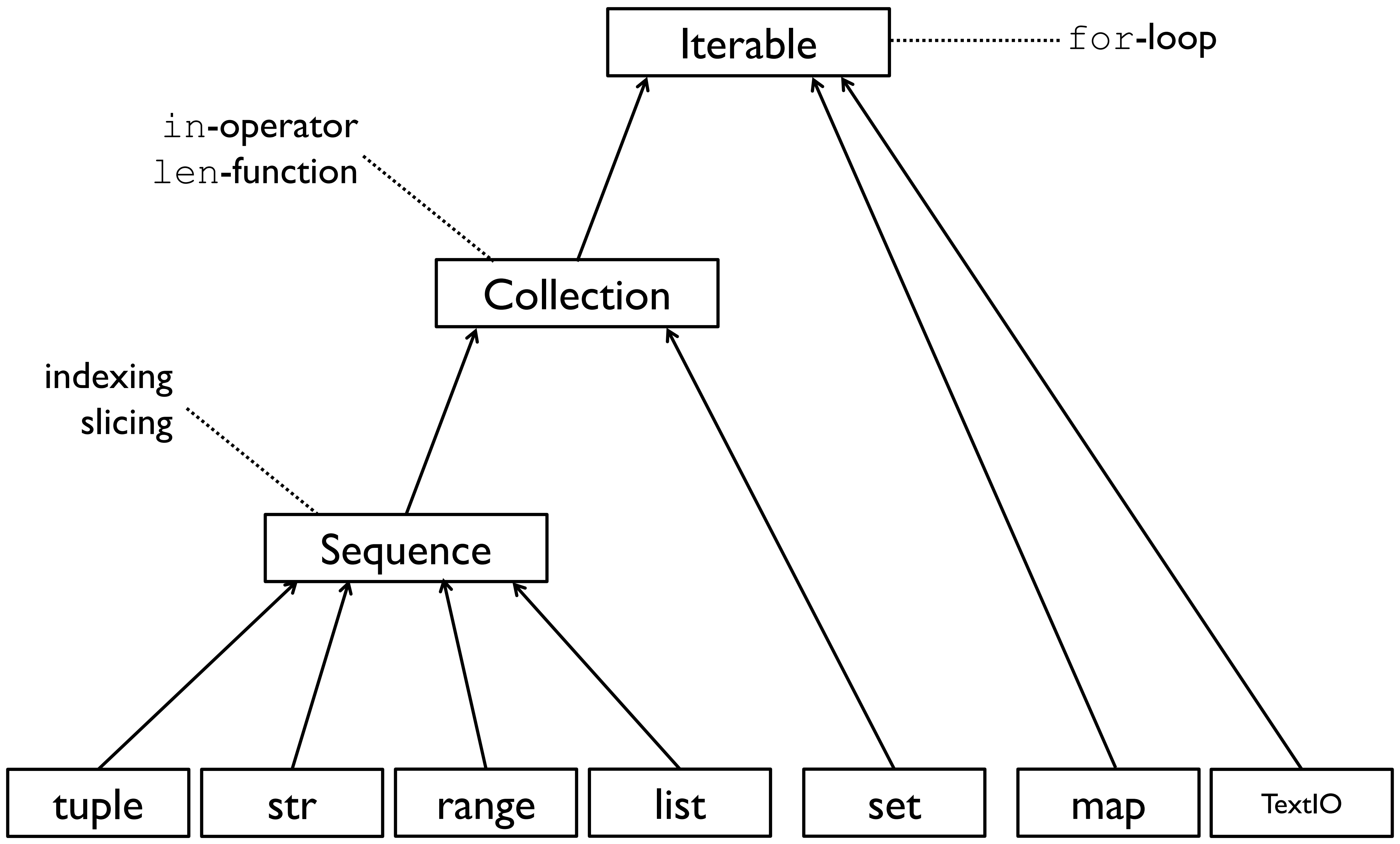


```
2
1
```

Python Tutor: https://goo.gl/soXX6e

- comma-separated expression evaluates to tuple
- tuples are immutable sequences
- multiple assignment assigns elements of sequence in parallel (evaluating first complete right-hand-side expression)

Iterable type hierarchy

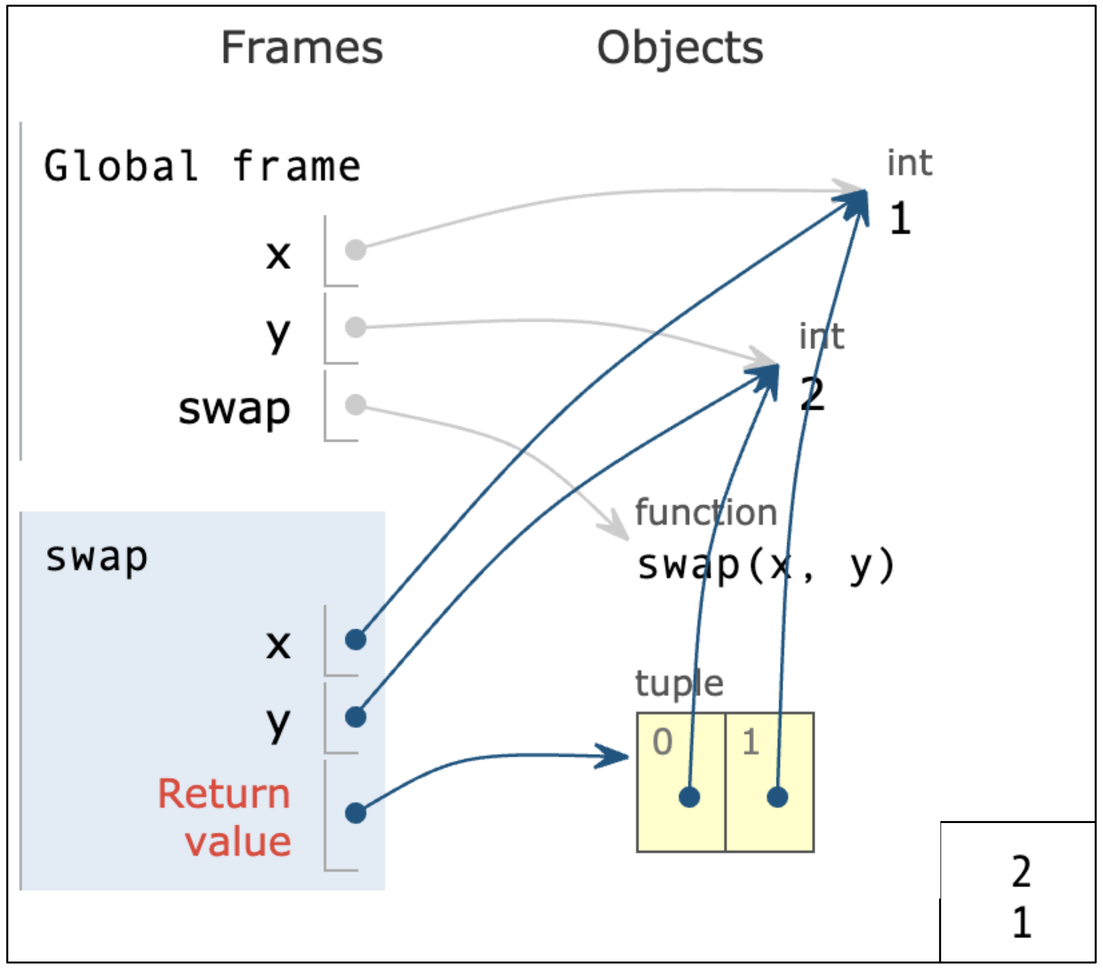


Can swap directly using tuple expression

```
x = 1
y = 2

def swap(x, y):
    return y, x

x, y = swap(x, y)
print(x)
print(y)
```



```
2
1
```

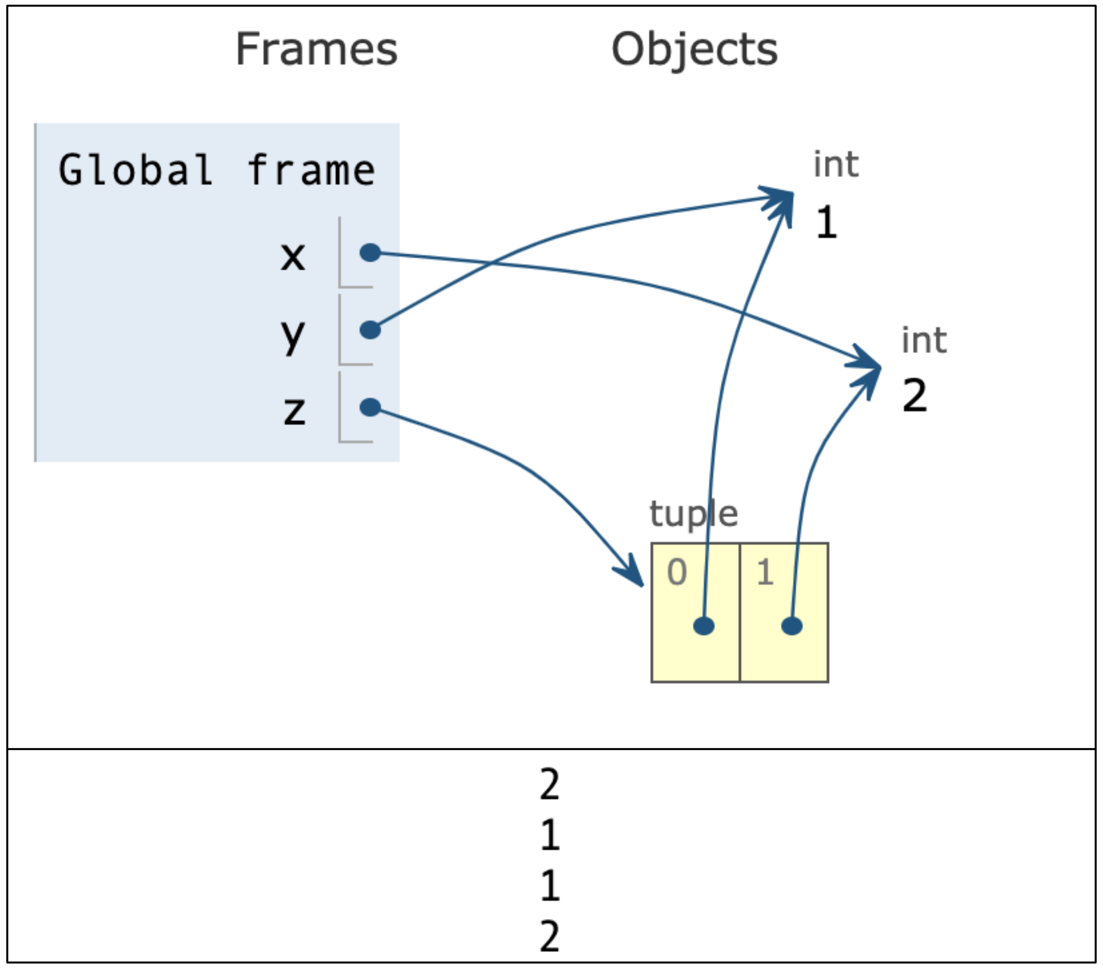
Python Tutor: https://goo.gl/BycpM5

- can directly write flipped tuple expression
- receive by multiple assignment

Simple enough to not need function

```
x = 1
y = 2
x, y = y, x
print(x)
print(y)

z = y, x
print(z[0])
print(z[1])
```



```
2
1
1
2
```

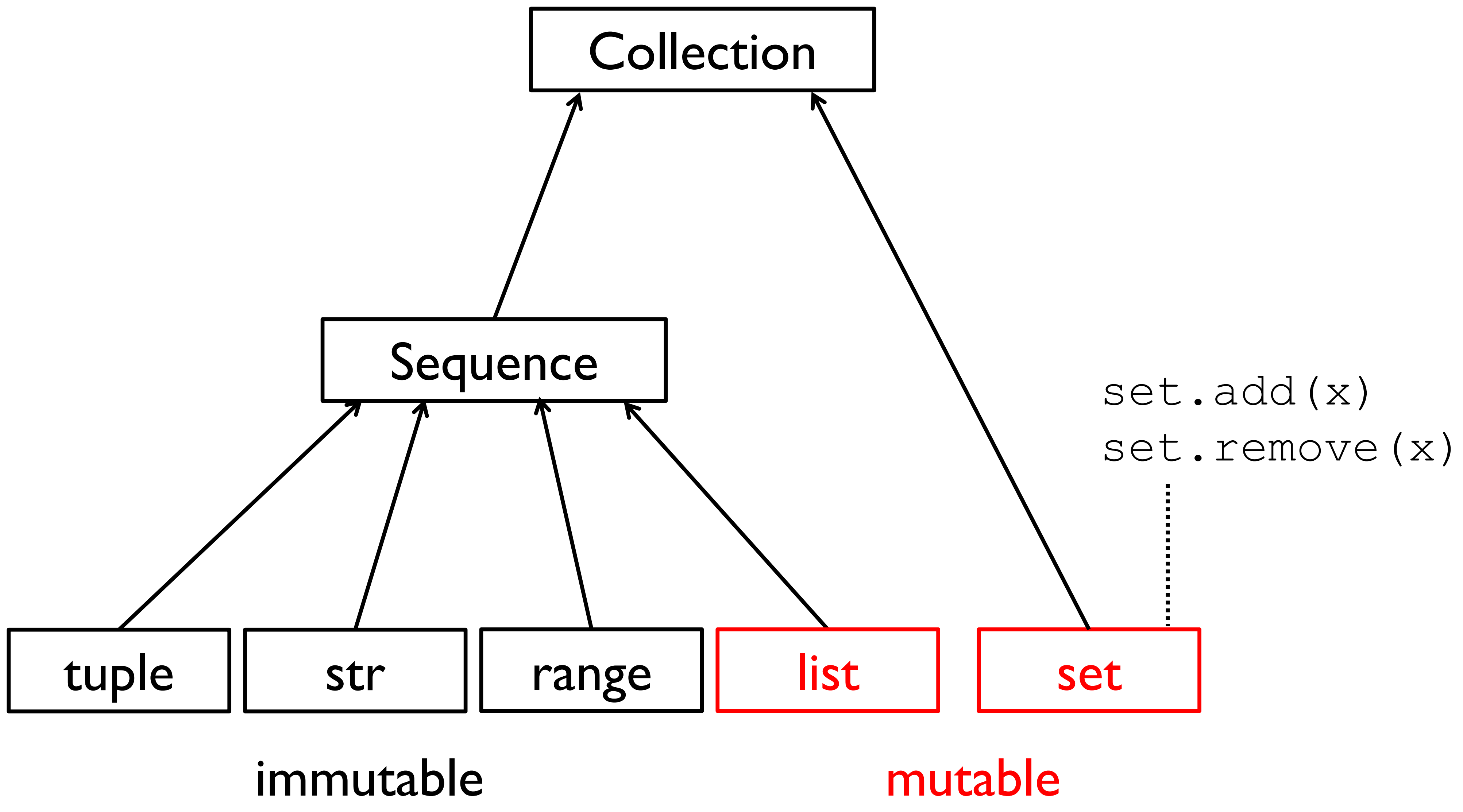
Python Tutor: https://goo.gl/9Ler1t

- can directly write flipped tuple expression
- receive by multiple assignment
- canonical way to swap references in Python
- sidenote: assigning to single variable yields tuple reference

Overview

1. Objects and variables: things and names for things
2. Functions, multiple assignments, and tuples
3. Mutability: the same and the similar

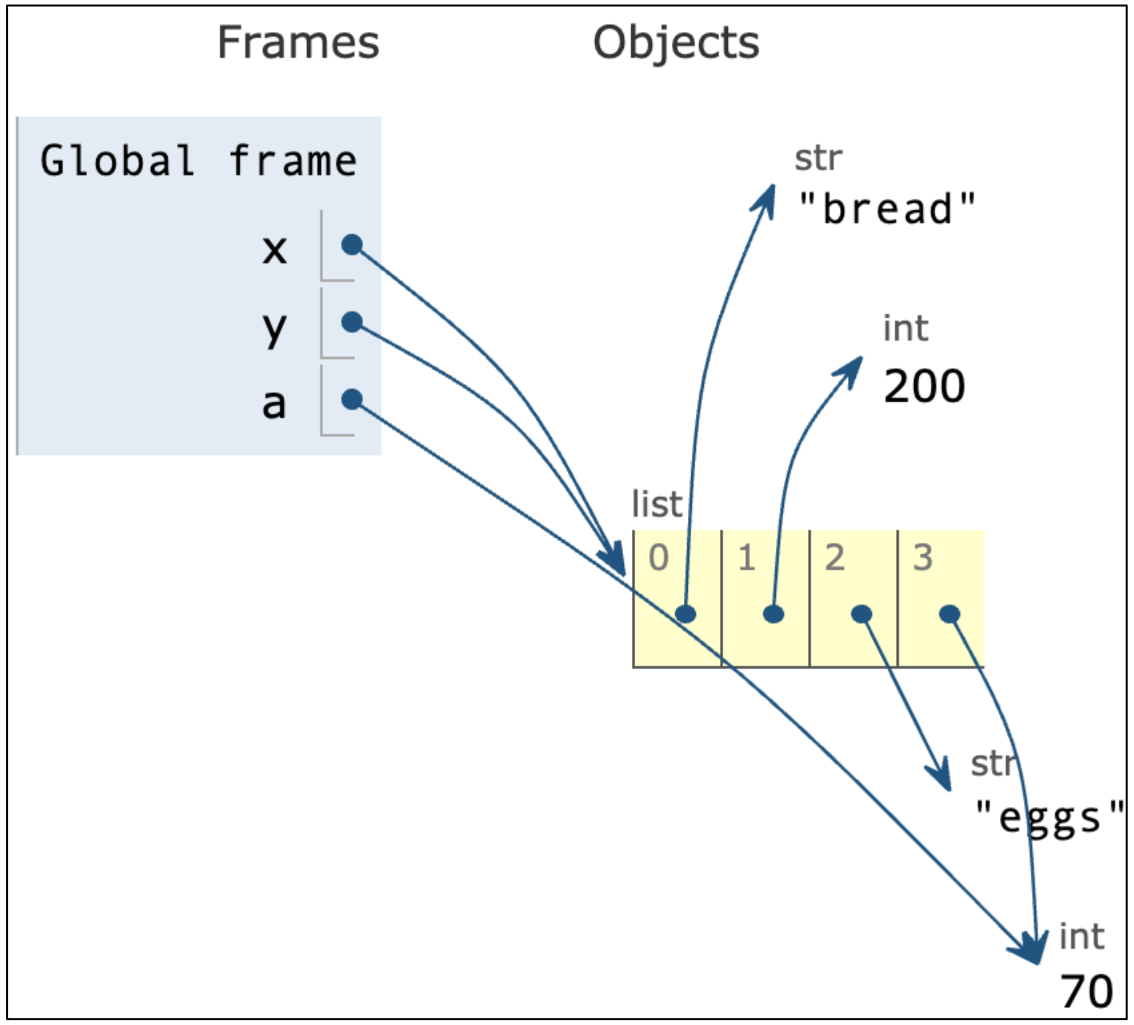
Mutable versus Immutable Collections



What are Python *lists*?

```
x = ['bread', 200]
y = x
x.extend(['eggs', 50])
x[-1] = 70

a = y[-1]
```



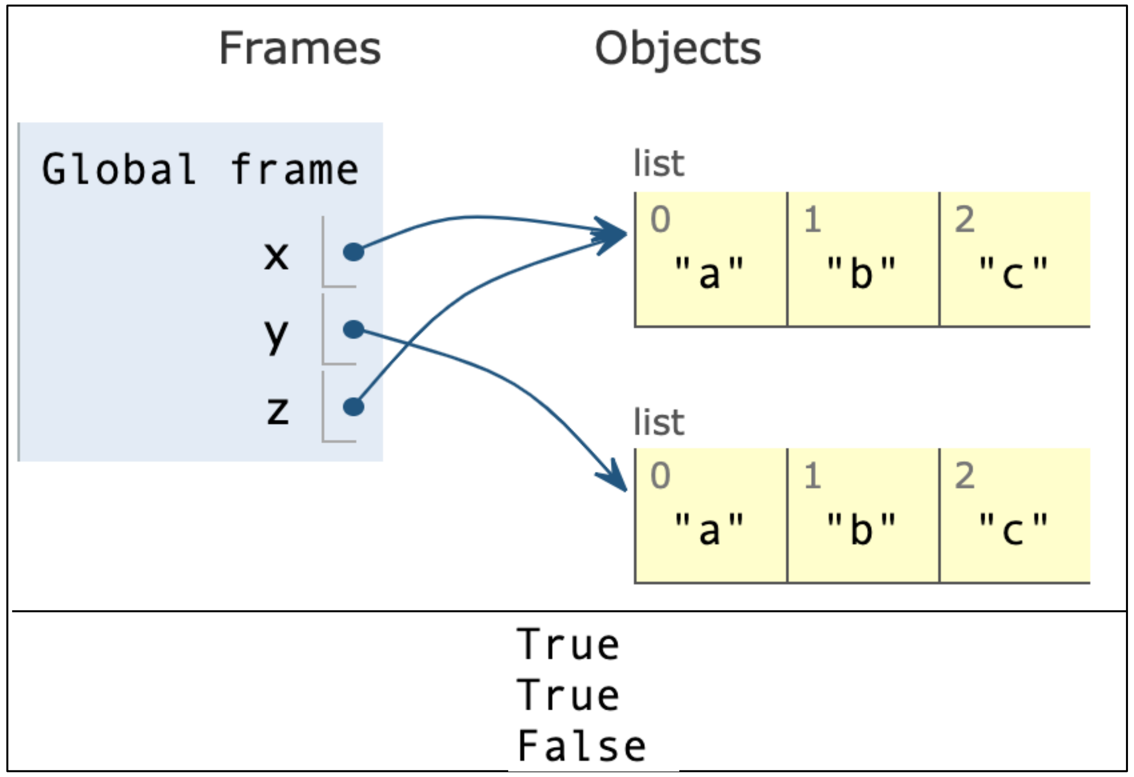
Python Tutor: https://goo.gl/EuExCB

- *lists* are objects containing mutable sequence of references
- values of variables referencing same list change simultaneously!

How are new lists created?

```
x = ['a', 'b', 'c']
y = ['a', 'b', 'c']
z = x

print(x==y and x==z)
print(id(x)==id(z))
print(id(x)==id(y))
```



```
True
True
False
```

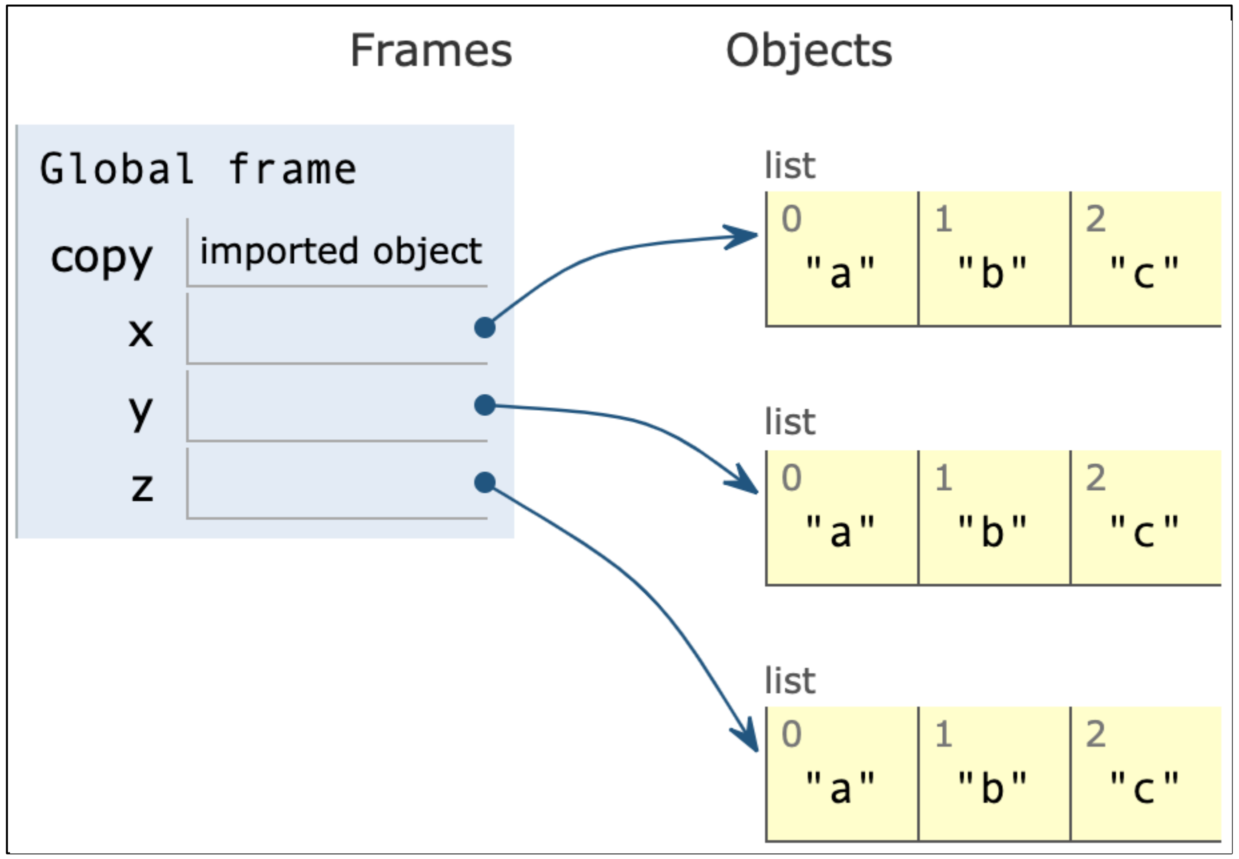
Python Tutor: https://goo.gl/J9J18h

- list objects are not generally re-used: "[…]"-expression evaluates to new list
- but: assignment assigns *reference* to *existing* object (as usual)
- same objects are always equal
- equal objects don't necessarily have same identity

How can we copy a list?

```
from copy import copy

x = ['a', 'b', 'c']
y = x[:]
z = copy(y)
```



Python Tutor: https://goo.gl/fAJXe5

- certain expressions result in copy of list (e.g., canonical way is "slicing" of complete original list)
- copy module provides function *copy* for all types of objects

How can we copy a list?

```
from copy import copy

t1 = [['a', 'b'],
      ['c', 'd']]

t2 = copy(t1)
t2[0][0] = 'A'

print(t1[0][0])
```

https://flux.qa

Clayton: AXXULH
Malaysia: LWERDE

- certain expressions result in copy of list (e.g., canonical way is "slicing" of complete original list)
- copy module provides function *copy* for all types of objects

How can we copy a list?

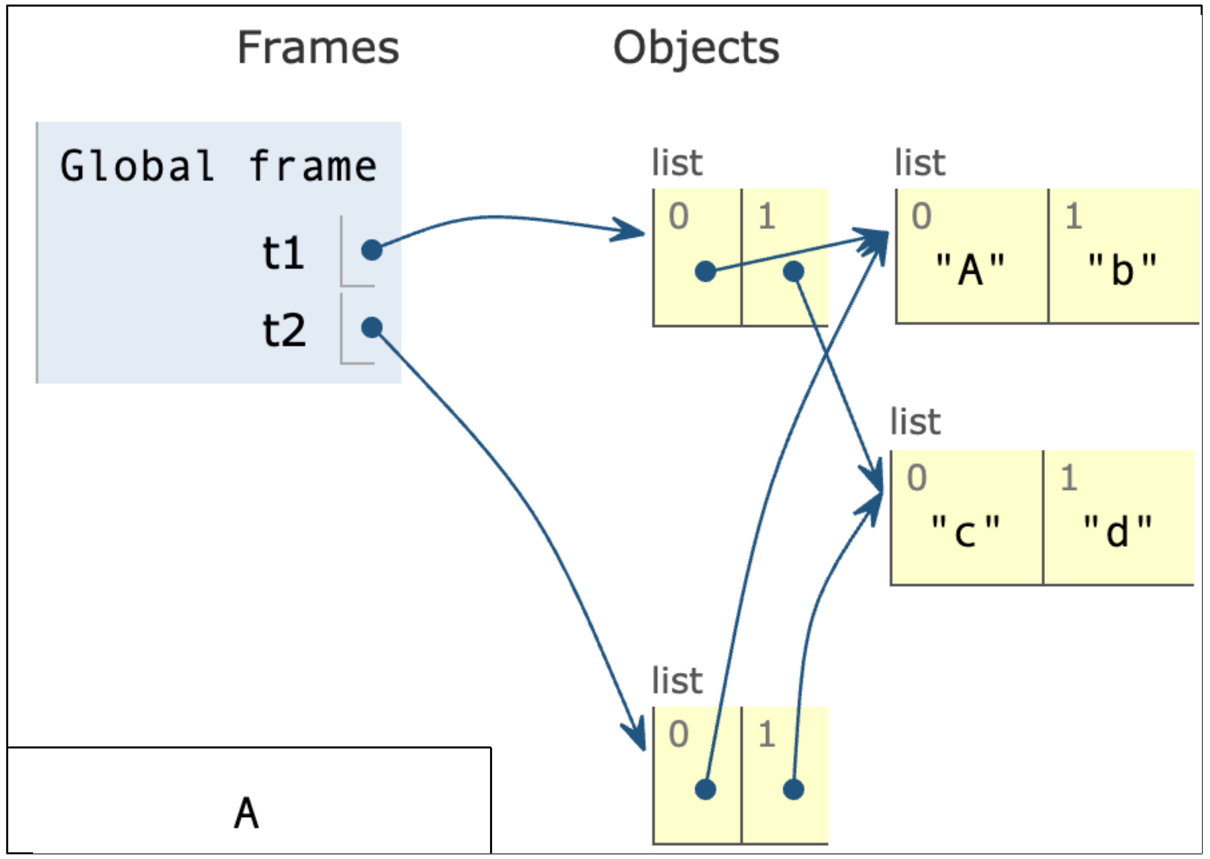
```
from copy import copy

t1 = [['a', 'b'],
      ['c', 'd']]

t2 = copy(t1)
t2[0][0] = 'A'

print(t1[0][0])
```

Python Tutor: https://goo.gl/VCnf9o



- certain expressions result in copy of list (e.g., canonical way is "slicing" of complete original list)
- copy module provides function *copy* for all types of objects
- however, these copies are *shallow*
- remember, list is sequence of references: simple copy only copies references and not object referenced

How can we create a "deep" copy?

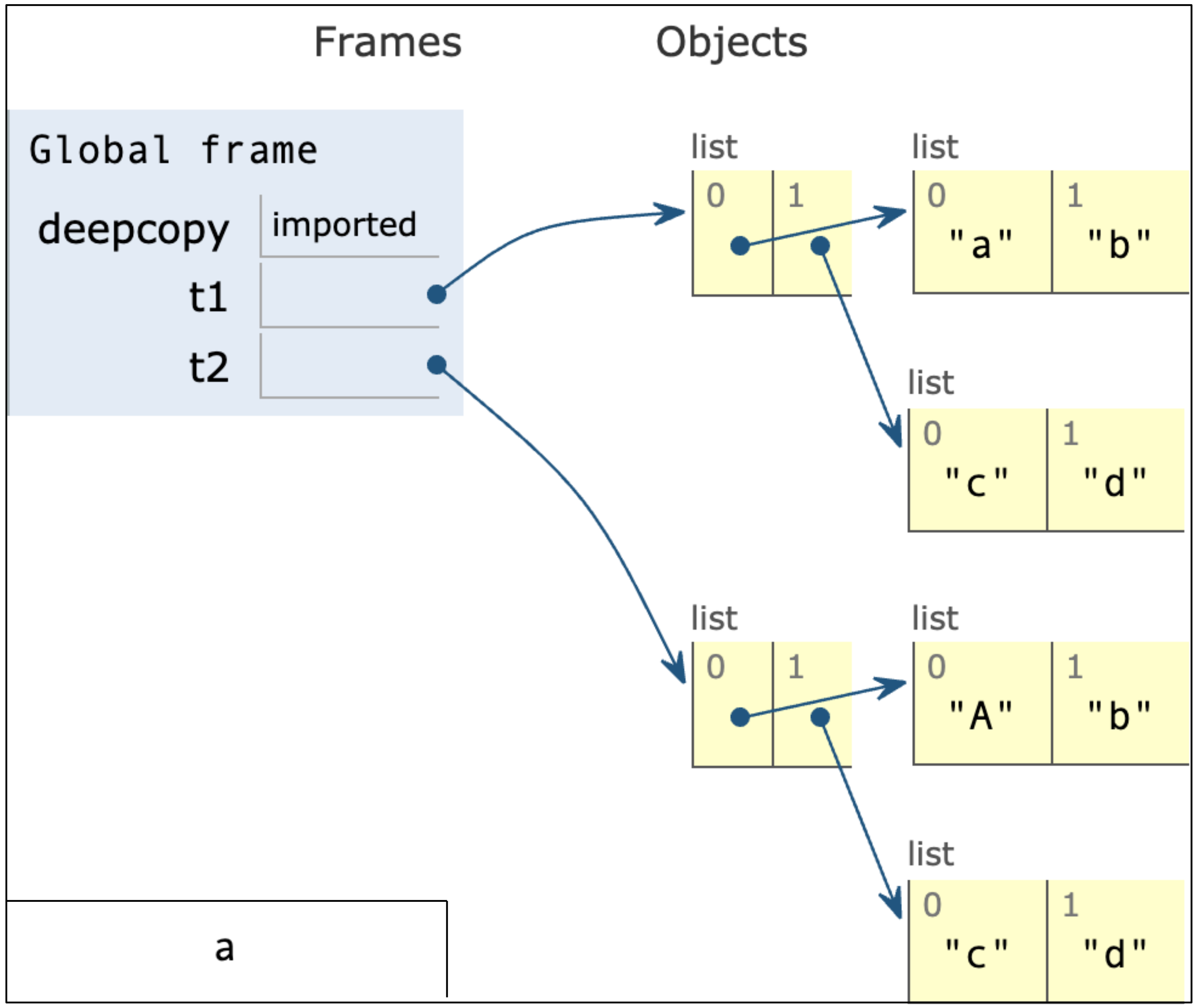
```
from copy import deepcopy

t1 = [['a', 'b'],
      ['c', 'd']]

t2 = deepcopy(t1)
t2[0][0] = 'A'

print(t1[0][0])
```

Python Tutor: https://goo.gl/2fQDap



- copy module provides function deepcopy that *recursively* copies referenced objects
- resulting deep copy is fully independent copy of original object

Augmented assignment statements

```
>>> x = 1
>>> x *= 2
>>> x
2
>>> x += 4
>>> x
6
>>> w = 'ab'
>>> w += 'c'
>>> w
'abc'
```

short-hand for `x = x * 2`

- short-hand for applying operator to *immutable* object referenced by variable and reassigning result

Augmented assignment statements with mutable objects

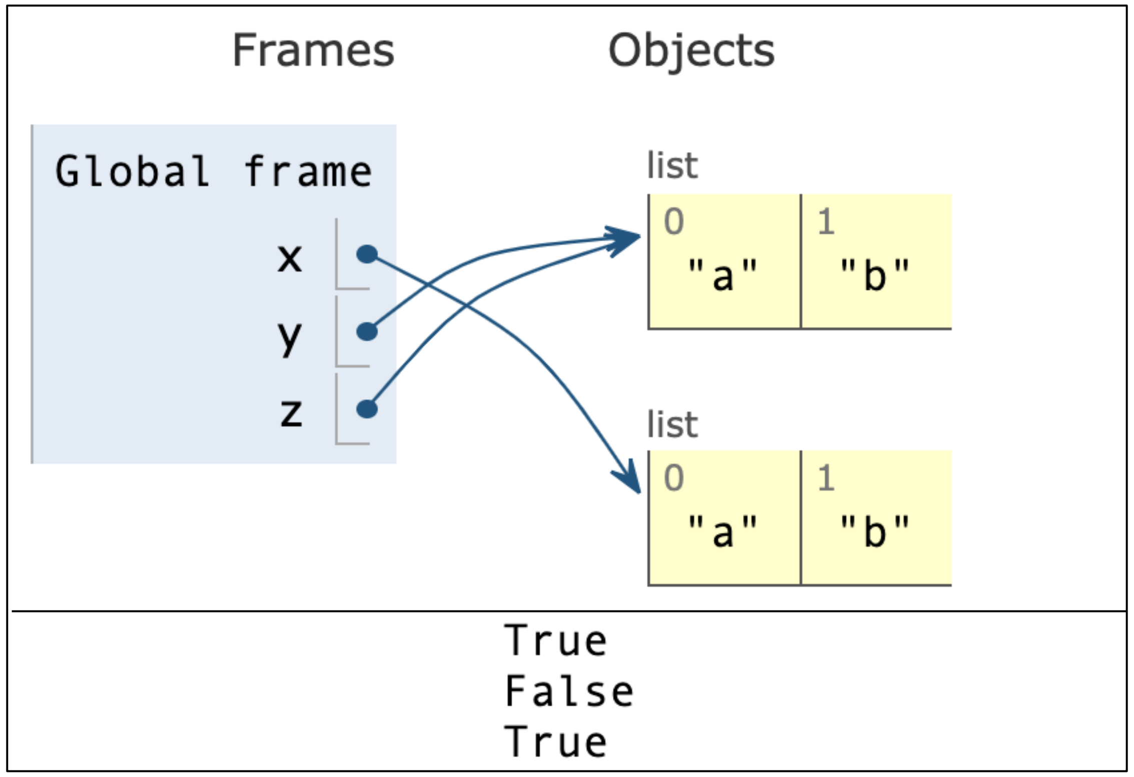
```
x = ['a']
y = x
z = y

print(x is y and y is z)

x = x + ['b']
y += ['b']

print(x is z)
print(y is z)
```

short-hand for `id(x)==id(y)`



```
True
False
True
```

Python Tutor: https://goo.gl/j2CrLu

- short-hand for applying operator to *immutable* object referenced by variable and reassigning result
- *not* equivalent for mutable objects: those are modified *in-place* instead of creating new object
- can be confusing; alternative: mutator methods (e.g. extend)

What have we learned

- Variables are named references to objects
  - can be re-assigned
  - loose object when no longer referenced
- Functions are objects, too
  - bound to variable on definition
  - references to them can be passed to other variables (including arguments of other functions)
- Tuples are immutable sequence types
  - can be used for multiple assignment
  - useful is function needs to return more than one value
- Mutable objects (e.g., lists) behave differently than immutable objects (ints, floats, strings, Booleans)
  - value of variable pointing to mutable object can be changed without being re-assigned
  - augmented assignment changes in-place

Before Next Lecture

Try out Python Tutor (e.g., examples of this lecture)

“Introduction to the design and analysis of algorithms”

Chapter 1

Coming Up

- Sorting
- Reasoning about algorithms (Invariants)