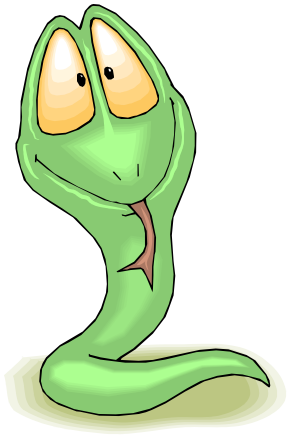
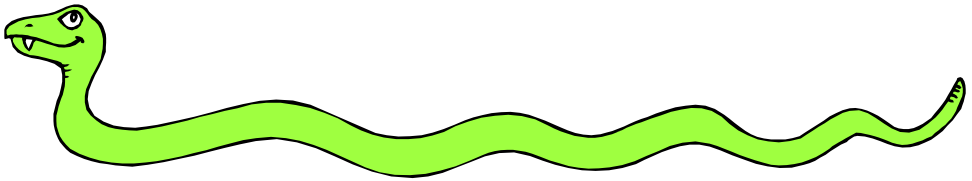
Intro to Python

- Dictionaries
- Functions
- Reference semantics



Dictionaries



Dictionaries: A *Mapping* type

- **Dictionaries store a *mapping* between a set of keys and a set of values.**
  - Keys can be any *immutable* type.
  - Values can be any type
  - A single dictionary can store values of different types
- **You can define, modify, view, lookup, and delete the key-value pairs in the dictionary.**

Creating and accessing dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user']
'bozo'

>>> d['pswd']
1234

>>> d['bozo']

Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo
```

Updating Dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}
```

- Keys must be unique.
- Assigning to an existing key replaces its value.

```
>>> d['id'] = 45
>>> d
{'user':'clown', 'id':45, 'pswd':1234}
```

- Dictionaries may be unordered depending on the version of Python used
  - Unless you are absolutely sure of the version of Python will be executing your code, it is perhaps best to avoid depending upon key-storage order...
- (Dictionaries work by *hashing*)

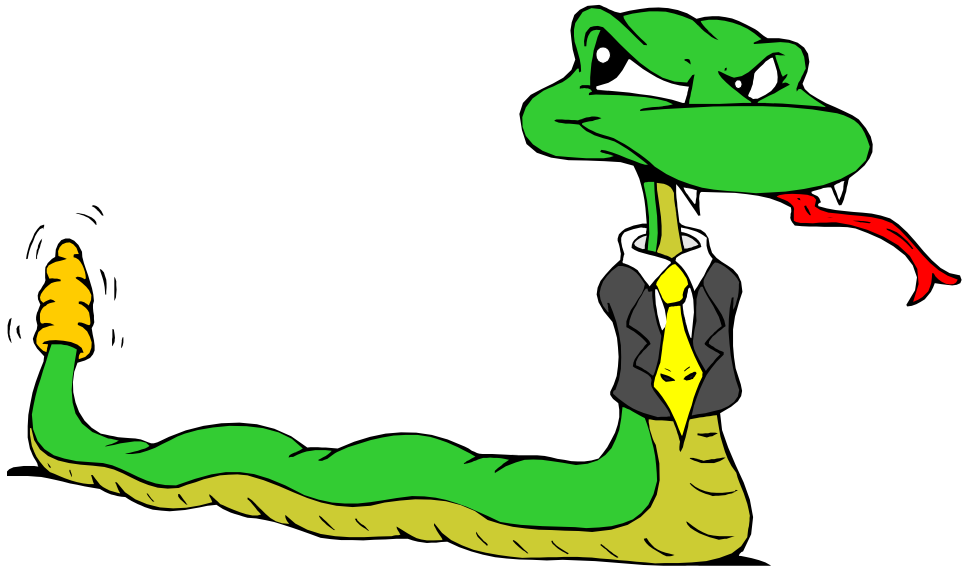
Removing dictionary entries

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> del d['user']          # Remove one key/value pair.
>>> d
{'p':1234, 'i':34}

>>> d.clear()              # Remove all key/value pairs.
>>> d
{}
```

Useful Accessor Methods

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> d.keys()                  # "List" of keys.
dict_keys(['user', 'p', 'i'])  # Order: YMMV

>>> list(d.keys())            #  Actual list
['user', 'p', 'i']

>>> d.values()                # List of values.
dict_values(['bozo', 1234, 34])

>>> d.items()          # List of item tuples.
dict_items([('user', 'bozo'), ('p',1234), ('i',34)])
```

Functions in Python



Defining Functions

Function definition begins with "def"

Function name and its arguments.

```
def get_final_answer(filename):
    """Documentation String"""
    line1
    line2
    return total_counter
```

Colon.

The indentation matters…
First line with less indentation is considered to be outside of the function definition.

The keyword 'return' indicates the value to be sent back to the caller.

No header file or declaration of types of function or arguments.

Reminder: Python and Types

Python determines the data types of *variable bindings* in a program automatically. ***"Dynamic Typing"***

But Python's not casual about types, it enforces the types of *objects*. ***"Strong Typing"***

So, for example, you can't just append an integer to a string. You must first convert the integer to a string itself.

```
x = "the answer is "  # Deduces x is bound to a string.
y = 23                # Deduces y is bound to an integer.
print (x + y)         # Python will complain about this.
```

Calling a Function

- **The syntax for a function call is:**

```
>>> def myfun(x, y):
        return x * y
>>> myfun(3, 4)
12
```

- **Parameters in Python are "Call by Assignment."**
  - Sometimes acts like "call by reference" and sometimes like "call by value" in C++.
    - Mutable datatypes: Behaves like Call-by-reference.
    - Immutable datatypes: Behaves like Call-by-value.

Functions without returns

- ***All* functions in Python have a return value**
  - even if no *return* line inside the code.
- **Functions without a *return* actually do return the special value *None*.**
  - *None* is a special constant in the language.
  - *None* is used like *NULL*, *void*, or *nil* in other languages.
  - *None* is also logically equivalent to False.
  - The interpreter doesn't print *None*

Function overloading? No.

- **There is no function overloading in Python.**
  - Unlike C++, a Python function is specified by its name alone
    - — The number, order, names, or types of its arguments cannot be used to distinguish between two functions with the same name.
  - Two different functions can′t have the same name, even if they have different arguments.
- **But: see *operator overloading* later in the course's treatment of Python**

Functions are first-class objects in Python

- Functions can be used as any other data type
- If something is a first-object, it can be:
  - an argument to a function
  - a return values from a function
  - assigned to a variable
  - a part of a tuple, list, or any other containers

```
>>> def myfun(x):
        return x*3

>>> def applier(q, x):
        return q(x)

>>> applier(myfun, 7)
21
```

Default Values for Arguments

- You can provide default values for a function's arguments
- These arguments are optional when the function is called

```
>>> def myfun(b, c=3, d="hello"):
            return b + c
>>> myfun(5,3,"hello")
>>> myfun(5,3)
>>> myfun(5)
```

Each function call above returns 8.

The Order of Arguments

- **You can call a function with some or all of its arguments out of order as long as you specify them (these are called keyword arguments). You can also just use keywords for a final subset of the arguments.**

```
>>> def myfun(a, b, c):
        return a-b
>>> myfun(2, 1, 43)
  1
>>> myfun(c=43, b=1, a=2)
  1
>>> myfun(2, c=43, b=1)
  1
```

A detour: command-line args

```
#!/usr/bin/env python

import sys

def main():
    if len(sys.argv) == 1:
        print ("No arguments")
    else:
        print ("First argument is", sys.argv[1])

if __name__ == "__main__":
    main()
```

We will eventually look at the meaning and forms of the **import** statement. For now, however, you consider it as having roughly the same meaning as in Java.

This approach to retrieving command-line arguments is a bit distasteful given we can use the argparse module.

However, sometimes you do want the raw arguments/options, and this approach will give them to you in the sys.argv array.

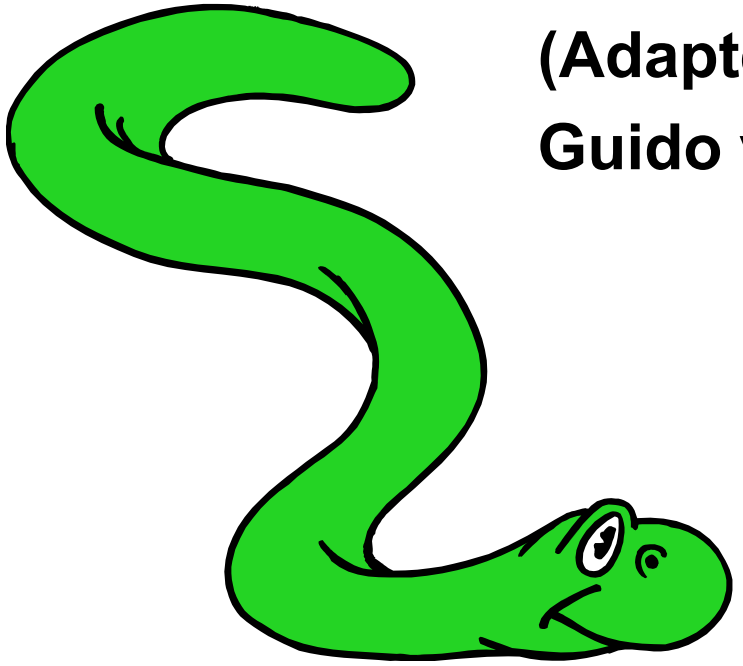
A detour: command-line args

```
#!/usr/bin/env python  # Don't need to guess bang path for "python"

import argparse

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--width', type=int, default=75,
        help='length of line')
    parser.add_argument('--indent',type=int, default=0,
        help='number of blank characters to use at start of line')
    parser.add_argument('--number', type=int, default=10,
        help="add line numbering")
    parser.add_argument('filename', nargs='?', help='file to be processed')

    args = parser.parse_args()

    print ('width: {}; indent: {}; number: {}'.format( args.width,
        args.indent, args.number))
    if args.filename:
        print ('filename: ', args.filename)
    else:
        print ('no filename specified')

if __name__ == "__main__":
    main()
```

Another detour: "main" function

```
#!/usr/bin/env python # Must be where Python is located...

def main():
    print ("Here we are in main. About to visit caveOfCaerbannog.")
    caveOfCaerbannog()
    print()
    print ("Now we're back in main. About to call camelot().")
    camelot()
    print()
    print ("I feel happy! I feel hap...")

def caveOfCaerbannog():
    print ("We are visiting the dreadful Cave of Caerbannog.")
    print ("Heck, there are cute rabbits here like at UVic.")
    print ("Come here little raaaaa... AUGH!")

def camelot():
    print ("Here we are in Camelot.")
    print ("Let's leave. It is a silly place.")

if __name__ == "__main__":
    main()
```

Understanding Reference Semantics in Python



(Adapted from several slides by Guido van Rossum)

Understanding Reference Semantics

- **Assignment manipulates references**

  x = y **does not make a copy** of the object y references

  x = y makes x **reference** the object y references

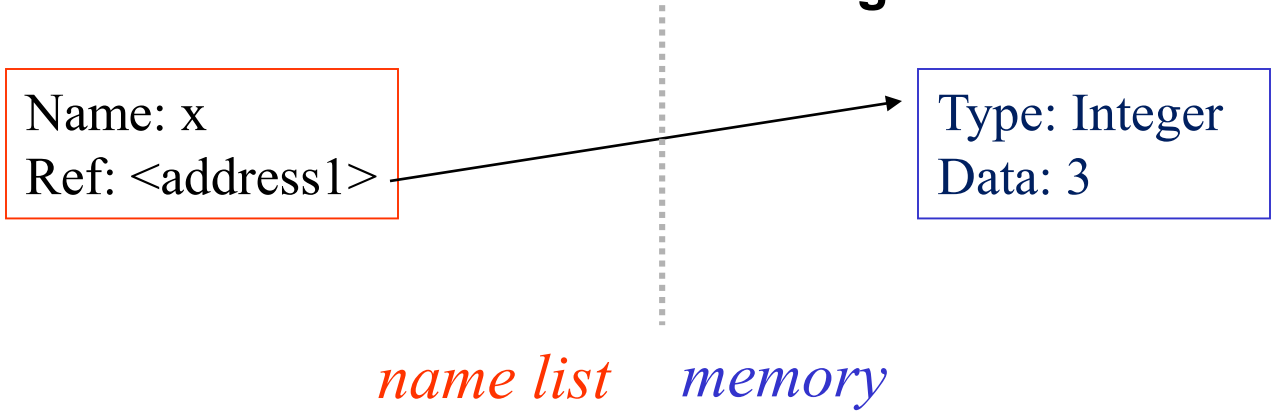
- **Very useful; but beware!**
- **Example:**

```
>>> a = [1, 2, 3]   # a now references the list [1, 2, 3]
>>> b = a           # b now references what a references
>>> a.append(4)     # this changes the list a references
>>> print(b)        # if we print what b references,
[1, 2, 3, 4]        # SURPRISE!  It has changed…
```

Why??

Understanding Reference Semantics II

- **There is a lot going on when we type:**
  `x = 3`
- **First, an integer *3* is created and stored in memory**
- **A name *x* is created**
- **A *reference* to the memory location storing the *3* is then assigned to the name *x***
- **So: When we say that the value of *x* is *3***
- **we mean that *x* now refers to the integer *3***



Understanding Reference Semantics III

- The data 3 we created is of type integer. In Python, the datatypes integer, float, and string (and tuple) are "immutable."
- This does not mean we cannot change the value of x, i.e. *change what x refers to* …
- For example, we could increment x:

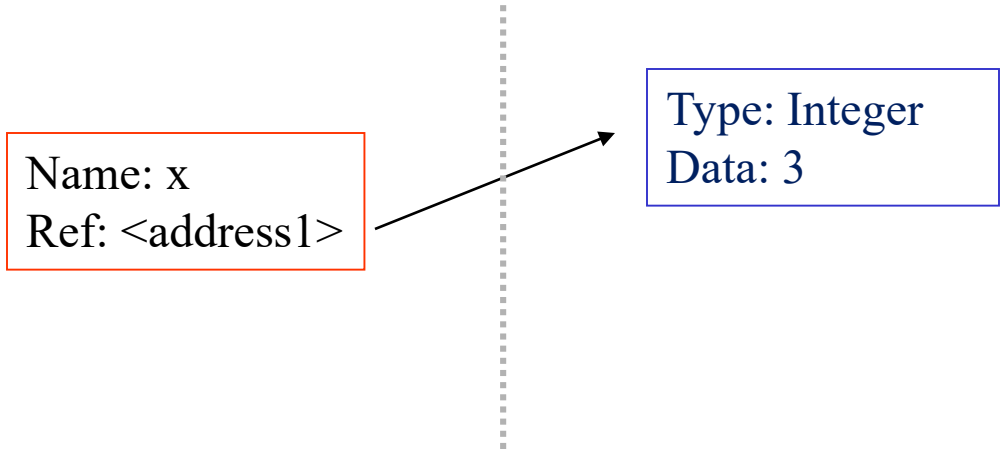
```
>>> x = 3
>>> x = x + 1
>>> print (x)
4
```

Understanding Reference Semantics IV

- **If we increment x, then what's really happening is:**

  1. *The reference of name **x** is looked up.*
  2. *The value at that reference is retrieved.*

```
>>> x = x + 1
```

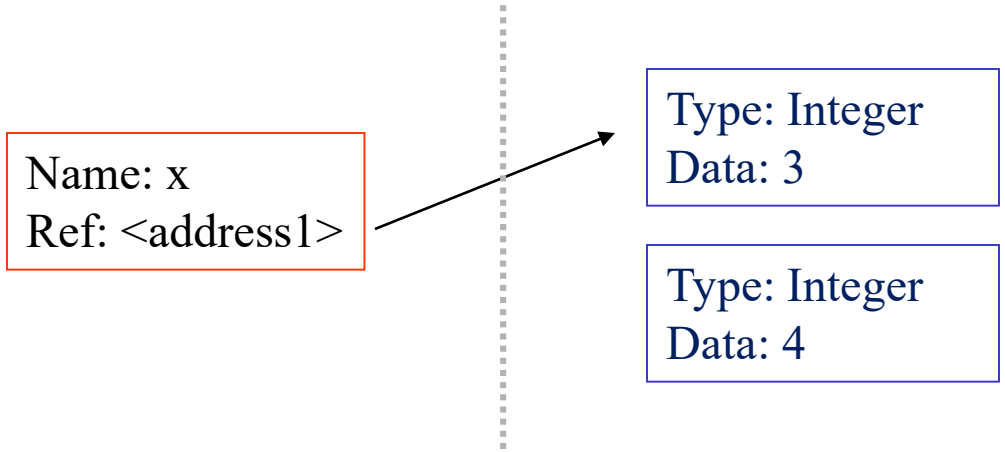


Understanding Reference Semantics IV

- **If we increment x, then what's really happening is:**

1. The reference of name ***x*** is looked up.
2. The value at that reference is retrieved.
3. *The 3+1 calculation occurs, producing a new data element* ***4*** *which is assigned to a fresh memory location with a new reference.*

```
>>> x = x + 1
```

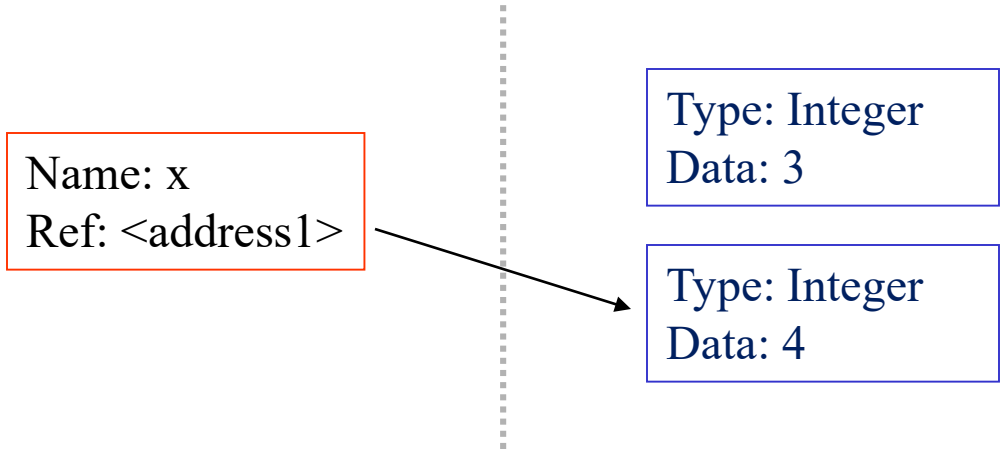


Understanding Reference Semantics IV

- **If we increment x, then what's really happening is:**

1. The reference of name ***X*** is looked up.
2. The value at that reference is retrieved.
3. The 3+1 calculation occurs, producing a new data element **4** which is assigned to a fresh memory location with a new reference.
4. *The name **X** is changed to point to this new reference.*

```
>>> x = x + 1
```

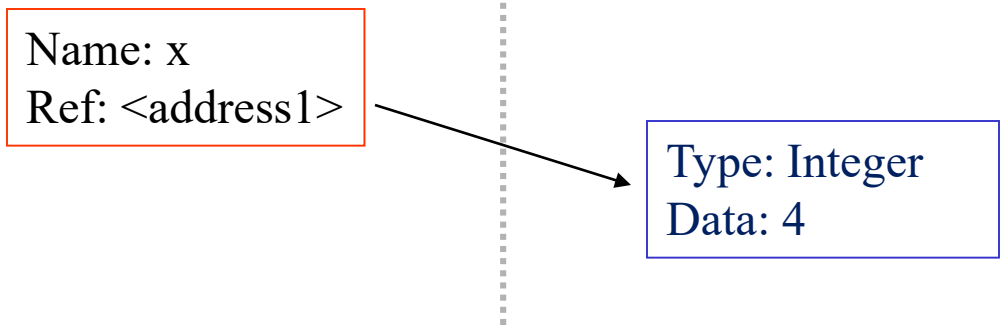


Understanding Reference Semantics IV

- **If we increment x, then what's really happening is:**

```
>>> x = x + 1
```

1. The reference of name ***x*** is looked up.
2. The value at that reference is retrieved.
3. The 3+1 calculation occurs, producing a new data element **4** which is assigned to a fresh memory location with a new reference.
4. The name **x** is changed to point to this new reference.
5. *The old data **3** is garbage collected if no name still refers to it.*



Assignment (part 1)

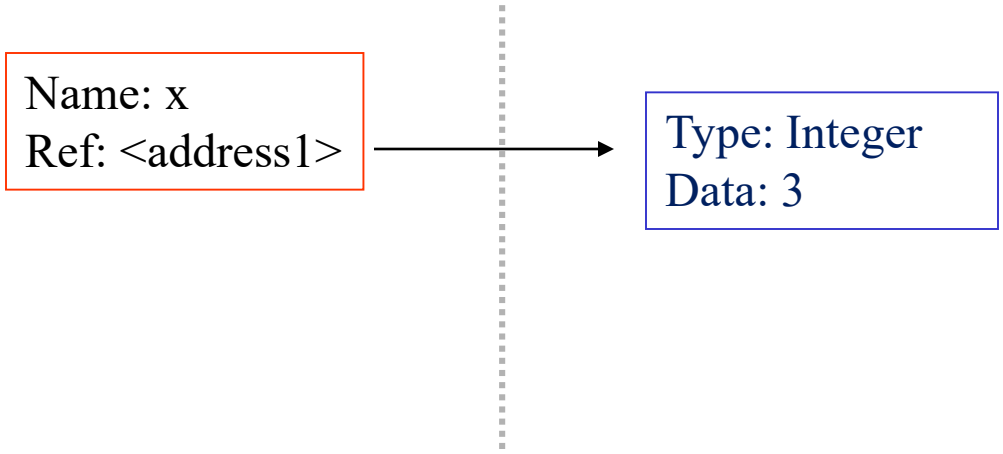
- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print(x)       # No effect on x, still ref 3.
3
```

Assignment (part 1)

- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

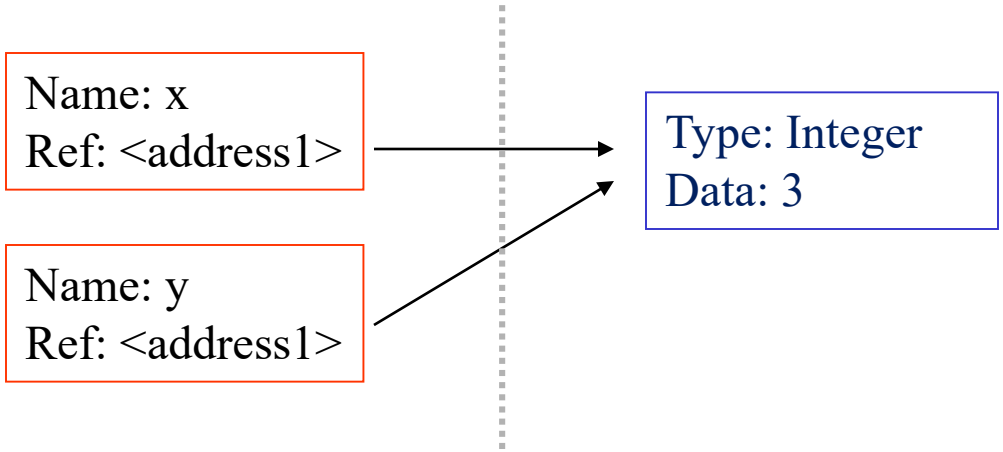
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print(x)       # No effect on x, still ref 3.
3
```



Assignment (part 1)

- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

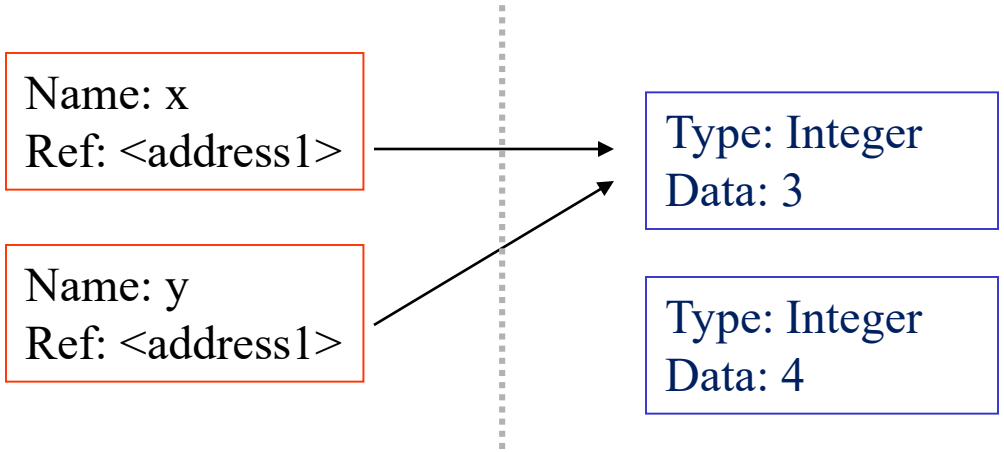
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print(x)       # No effect on x, still ref 3.
3
```



Assignment (part 1)

- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

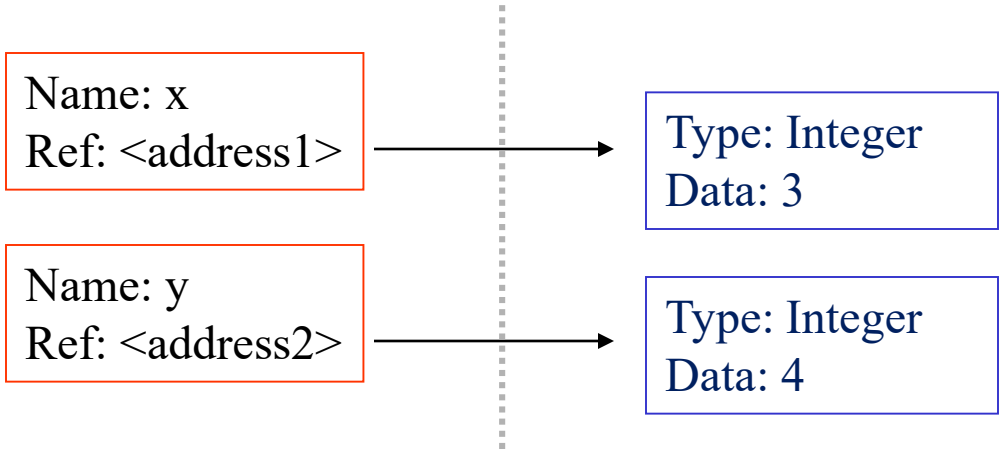
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print(x)       # No effect on x, still ref 3.
3
```



Assignment (part 1)

- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

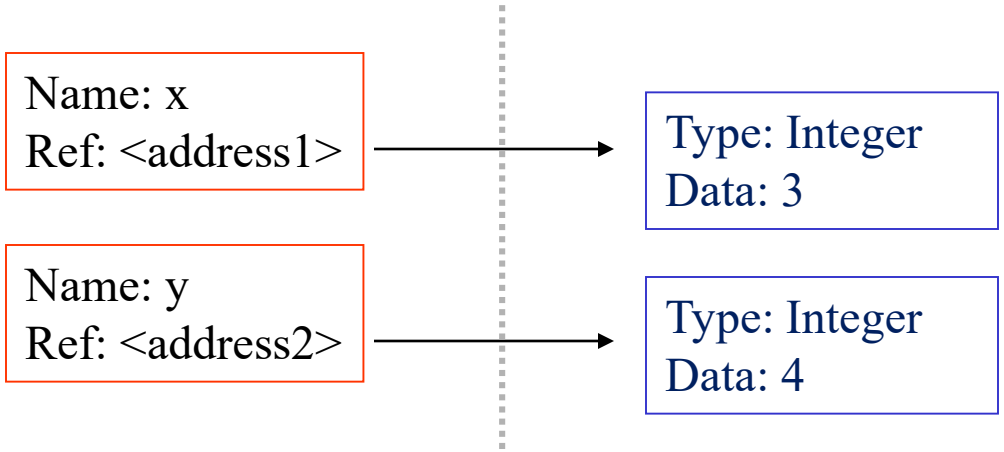
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print(x)       # No effect on x, still ref 3.
3
```



Assignment (part 1)

- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print(x)       # No effect on x, still ref 3.
3
```



Assignment (part 2)

- **For some other data types (lists, dictionaries, user-defined types), assignment works differently.**
  - These datatypes are **"mutable."**
  - When we change these data, we do it *in place.*
  - We don't copy them into a new memory address each time.
  - If we type y=x and then modify y, both x and y are changed.

immutable

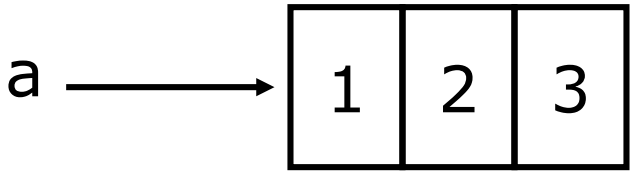
```
>>> x = 3
>>> y = x
>>> y = 4
>>> print x
3
```

mutable

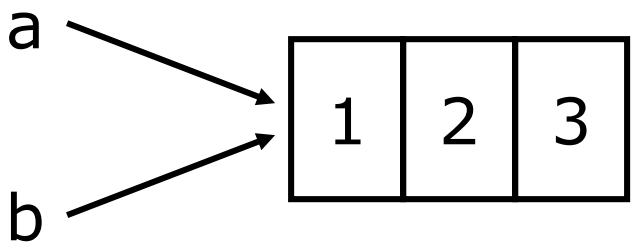
```
x = some mutable object
y = x
make a change to y
look at x
x will be changed as well
```

Why? Changing a Shared List

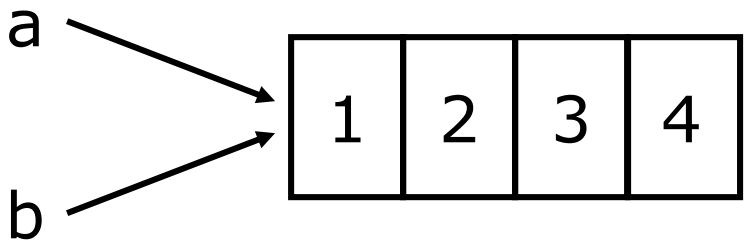
a = [1, 2, 3]



b = a



a.append(4)



Our surprising example surprising no more...

- **So now, here's our code:**

```
>>> a = [1, 2, 3]   # a now references the list [1, 2, 3]
>>> b = a           # b now references what a references
>>> a.append(4)     # this changes the list a references
>>> print (b)       # if we print what b references,
[1, 2, 3, 4]        # SURPRISE!  It has changed…
```