EPL451: Data Mining on the Web – Lab 4

University of Cyprus
Department of Computer Science

Παύλος Αντωνίου

Γραφείο: B109, ΘΕΕ01

Python

- Open source general-purpose language
- Object Oriented, Procedural, Functional
- Easy to interface with C/ObjC/Java/Fortran
- Easy-ish to interface with C++ (via SWIG)
- Great interactive environment (python idle)

- Downloads: http://www.python.org
- Documentation: http://www.python.org/doc/
- Free book: http://www.diveintopython.org
- **Powerful enterprise-ready open data-science platform: Anaconda https://www.continuum.io**

Install Anaconda3 with Python 3.6 on Linux

1. Open terminal
2. Run cd to move to HOME directory
3. wget https://repo.continuum.io/archive/Anaconda3-4.3.0-Linux-x86.sh
4. sudo bash Anaconda3-4.3.0-Linux-x86.sh

NOTE: Accept the default location or select a user-writable install location such as ~/anaconda.

Follow the prompts on the installer screens, and if unsure about any setting, simply accept the defaults, as they can all be changed later.

NOTE: If you select the option to not add the Anaconda directory to your bash shell PATH environment variable, you may later add this line to the file .bashrc in your home directory: export PATH="/home/username/anaconda/bin:$PATH" Replace /home/username/anaconda with your actual path.

The output of a successful installation will include the messages "Installation finished." and "Thank you for installing Anaconda!"

Finally, close and re-open your terminal window for the changes to take effect.

Install Anaconda3 with Python 3.6 on Windows

- Go to https://www.continuum.io/downloads#windows
- Download 64-bit or 32-bit installer depending on your machine architecture
- Double-click the **.exe** file to install Anaconda and follow the instructions on the screen

The Python Interpreter

- Interactive interface to Python

```
csdeptucy@ubuntu:~$ python3
Python 3.4.3 (default, Oct 14 2015, 20:33:09)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

- Python interpreter evaluates inputs:

```
>>> 3*(7+2) 27
```

- Python prompts with ‘>>>’.
- To exit Python:

```
>>>exit()
```

Running Programs on UNIX

- `csdeptucy@ubuntu:~$` **`python3 filename.py`**
- You should make the *.py file executable and add the following #!/usr/bin/env python3 to the top to make it runnable.
- Let’s go through the basics!

A Code Sample

```
x = 34 - 23                    # A comment.
y = "Hello"                    # Another one.
z = 3.45
if z == 3.45 or y == "Hello":        # : is needed
     x = x + 1                 # similar to x += 1.
     y = y + " World"          # String concatention.
print(x)                       # 12
print(y)                       # Hello World
x = y
print(x)                       # Hello World
```

Enough to Understand the Code

- Assignment uses `=` and comparison uses `==`
- For numbers `+ - * / %` are as expected
  - Special use of `+` for string concatenation
  - Special use of `%` for string formatting (as with printf in C)
- Logical operators are words (`and`, `or`, `not`) not symbols
- The basic printing command is `print`
- The first assignment to a variable creates it
  - Variable types don’t need to be declared
  - Python figures out the variable types on its own

Numeric Datatypes

- Integer numbers (int)
  - `z1 = 23`
  - `z2 = 5 // 2      # Answer is 2, integer division.`
  - `z3 = int(6.7)    # Converts 6.7 to integer. Answer is 6.`
  - Booleans (bool) are a subtype of integers
- Floating (float) point numbers (implemented using double in C)
  - `x1 = 3.456`
  - `x2 = 5 / 2       # Answer is 2.5`
- Complex numbers
- Additional numeric types: fractions, decimal
- See more here

Newlines and Whitespaces

- Use a newline to end a line of code.
  - Use \ when must go to next line prematurely.
- Whitespace is meaningful in Python: especially indentation
- No braces { } to mark blocks of code in Python…
  Use consistent indentation – whitespace(s) or tab(s) – instead.
  - The first line with more indentation starts a nested block
  - The first line with less indentation is outside of the block
  - Indentation levels must be equal within the same block
- Often a colon appears at the start of a new block.
  - e.g. in the beginning of `if`, `else`, `for`, `while`, as well as function and class definitions

```
if x%2 == 0:
   print("even")
   print("number")
else:
 print("odd")
```

Comments

- Single line comments: Start comments with # – the rest of line is ignored.
- Multiple line comments: Start/end comments with """
- Can include a “documentation string” as the first line of any new function or class that you define.
- The development environment, debugger, and other tools use it: it’s good style to include one.

```
def my_function(x, y):
    """This is the docstring. This
    function does blah blah blah."""
    # The code would go here...
```

Assignment

- Declaring a variable in Python means setting a **name** to hold a **reference** to some **object**.
  - Assignment creates references, not copies
- You create a name the first time it appears on the left side of an assignment expression:

```
x = 3  # x is the name of the reference to 3
```

- **Names** in Python **do not have** an intrinsic **type**. **Objects have types**.
  - Python determines the type of the reference automatically based on the data object assigned to it.
- A reference is deleted via garbage collection after any names bound to it have passed out of scope.

Accessing Non-Existent Names

- If you try to access a name before it's been properly created (by placing it on the left side of an assignment), you'll get an error.

```
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
>>> y = 3
>>> print(y)
3
```

- Multiple assignment

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

```
bob Bob _bob _2_bob_ bob_2 BoB
```

- There are some reserved words:

```
and, assert, break, class, continue, def, del,
elif, else, except, exec, finally, for, from,
global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while
```

Understanding Reference Semantics

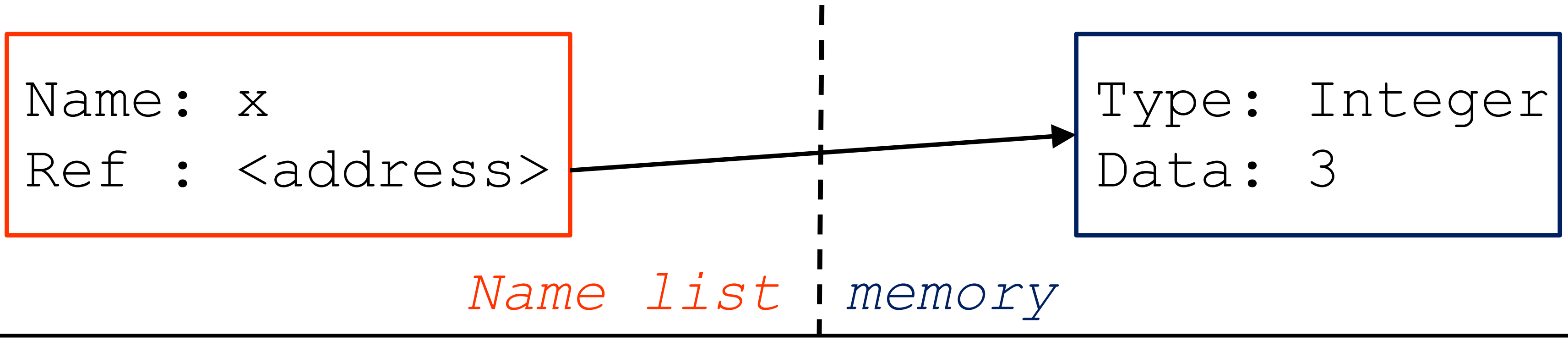
- Assignment manipulates references
  - `x = y` **does not make a copy** of the object y references
  - `x = y` makes x **reference** the object y references
- Very useful; but beware!
- Example:

```
>>> a = [1, 2, 3] # name a 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
- An **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

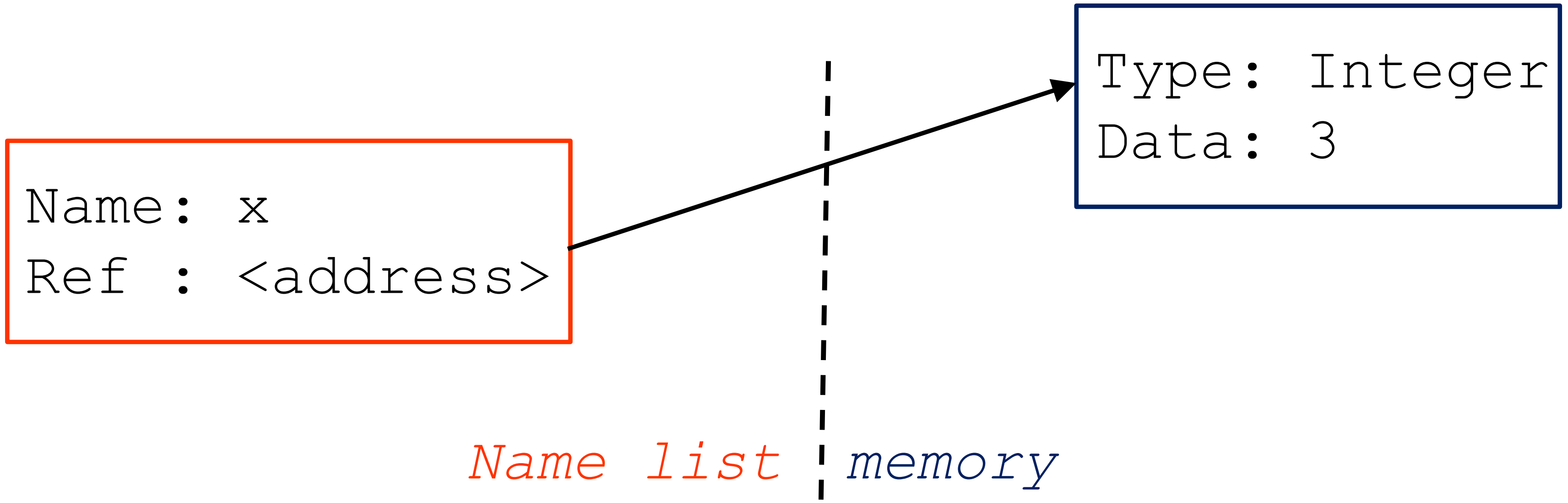
- The data 3 we created is of type integer.
- **In Python, the numeric datatypes (such as integers, floats) are "immutable" (αμετάβλητοι)**
- This doesn't mean we can't change the value of x, i.e. change what x refers to …
- For example, we could increment x:

```
>>> 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.

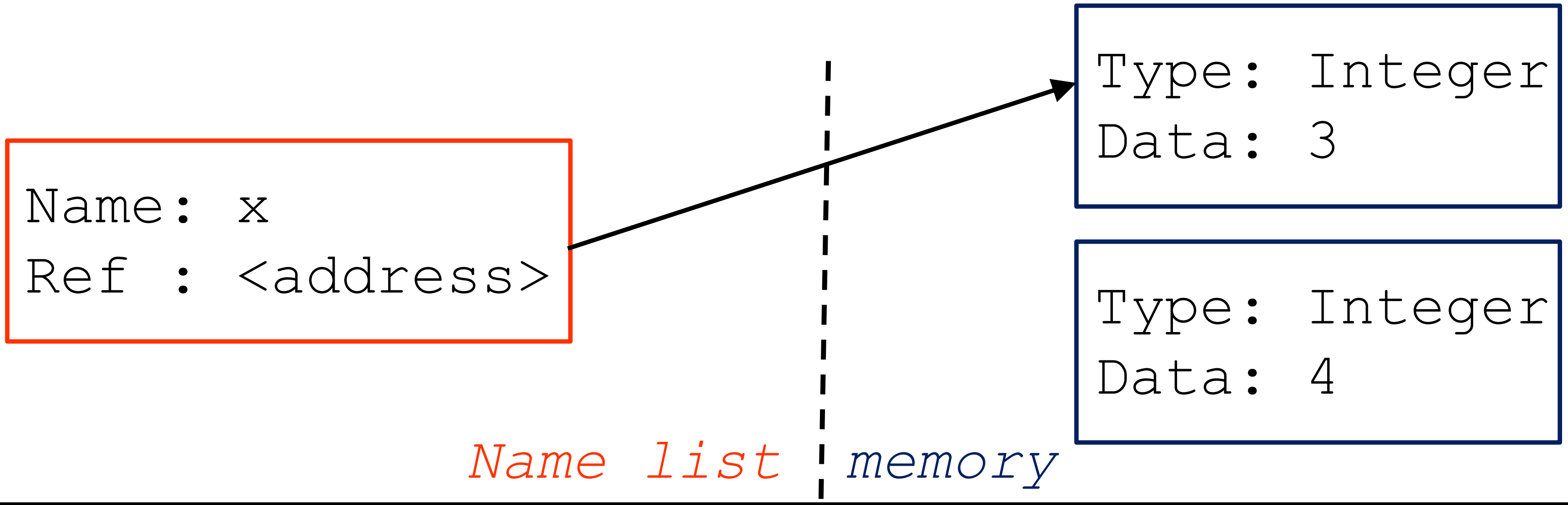
```
>>> x = x + 1
```



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.

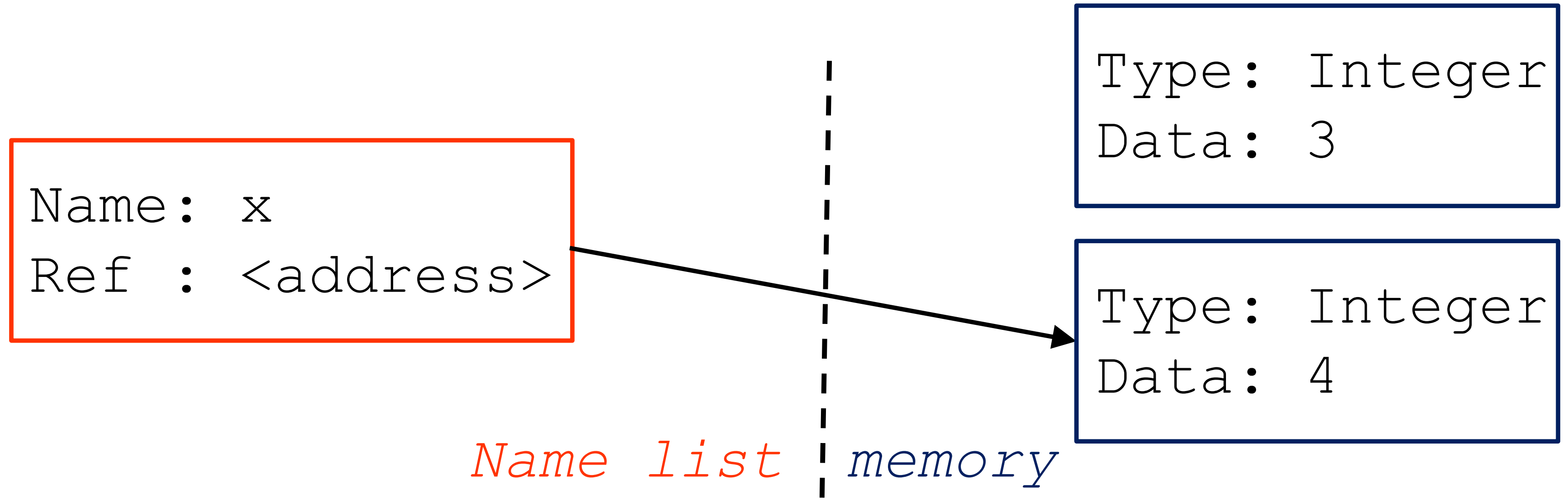
```
>>> x = x + 1
```



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.

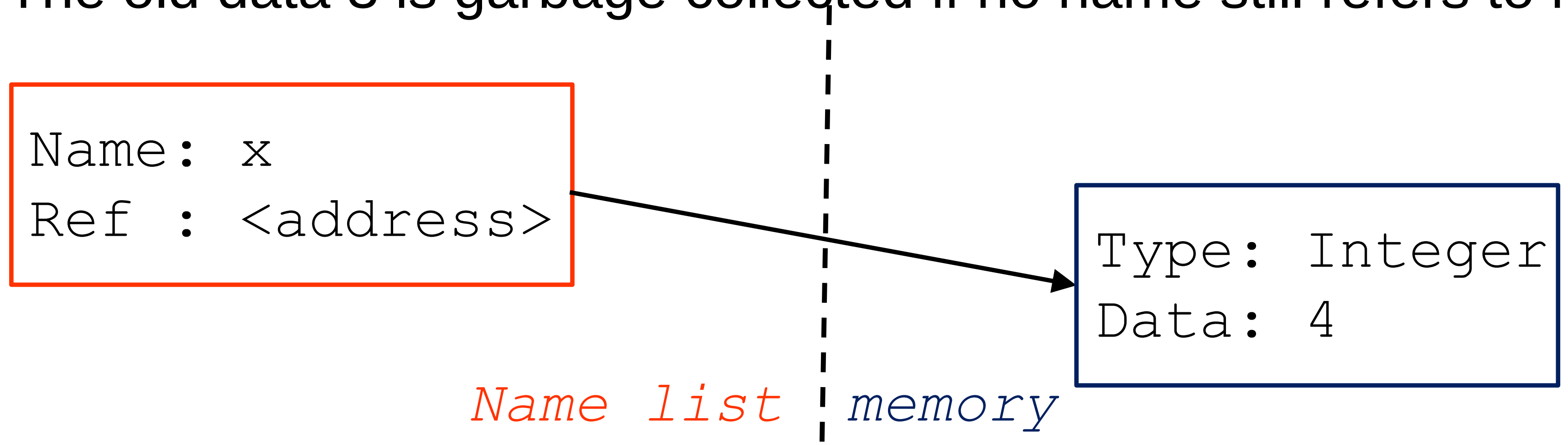
```
>>> x = x + 1
```



Understanding Reference Semantics IV

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

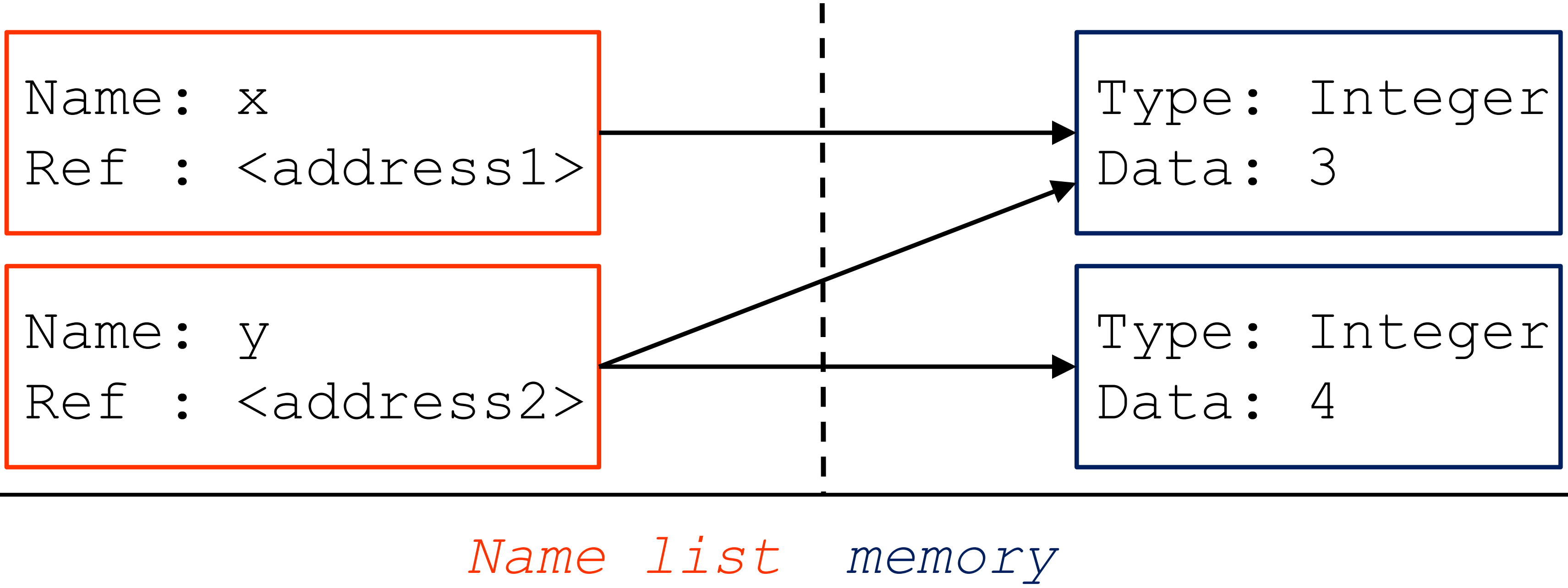
```
>>> x = x + 1
```



Understanding Reference Semantics V

- So, for numeric datatypes (integers, floats), 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
```



Some Python datatypes (objects)

- Some **immutable** objects
  - **int**
  - **float**
  - decimal
  - complex
  - bool

  Numeric datatypes
  - **string**
  - **tuple**
  - bytes
  - **range**

  Sequences
  - frozenset

  Set type

- Some **mutable** objects
  - **list**
  - bytearray

  Sequences
  - **set**

  Set type
  - **dict**

  Mapping
  - user-defined classes (unless specifically made immutable)

❖ When we change these data, this is done in place.
❖ They are not copied into a new memory address each time.

Immutable Sequences I

- **Strings**
  - Defined using double quotes `""` or single quotes `''`

```
>>> st = "abc"
>>> st = 'abc'
```
(Same thing.)

  - Unmatched can occur within the string.

```
>>> st = "matt's"
```

  - Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them:

```
>>> st = """This is a multi-line
string that uses triple quotes."""
>>> st = """a'b"c"""
```

Immutable Sequences II

- **Tuples (Πλειάδες)**
  - A simple **immutable** ordered sequence of items of mixed types
  - Defined using parentheses (and commas) or using `tuple()`.

```
>>> t = tuple()             # create empty tuple
>>> tu = (23, 'abc', 4.56, (2,3), 'def')   # another tuple
>>> tu[2] = 3.14
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

  - You can't change a tuple.
  - You can make a fresh tuple and assign its reference to a previously used name.

```
>>> tu = (23, 'abc', 3.14, (2,3), 'def')
```

Immutable Sequences III : data access

- We can access individual members of a **tuple** or **string** using square bracket “array” notation.
- Positive index: count from the left, starting with 0.
- Negative index: count from right, starting with –1.

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1] # Second item in the tuple.
'abc'
>>> tu[-3]
4.56
>>> st = "Hello World"
>>> st[1] # Second character in string.
'e'
```

Mutable Sequences I

- Lists
  - **Mutable** ordered sequence of items of mixed types
  - Defined using square brackets (and commas) or using `list()`.

```
>>> li = ["abc", 34, 4.34, 23]
```

  - We can access individual members of a list using square bracket “array” notation as in tuples and strings.

```
>>> li[1] # Second item in the list.
34
```

  - We can change lists in place.
    - Name li still points to the same memory reference when we’re done.
    - The mutability of lists means that they aren’t as fast as tuples.

```
>>> li[1] = 45
>>> li
['abc', 45, 4.34, 23]
```

Tuples vs. Lists

- Lists slower but more powerful than tuples.
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.
- To convert between tuples and lists use the list() and tuple() functions:

```
li = list(tu)
tu = tuple(li)
```

Slicing in Sequences: Return Copy of a Subset 1

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Return a **copy** of the container with a subset of the original members. Start copying at the first index, and stop copying before the second index.

```
>>> tu[1:4]
('abc', 4.56, (2,3))
```

- You can also use negative indices when slicing.

```
>>> tu[1:-1]
('abc', 4.56, (2,3))
```

Slicing in Sequences: Return Copy of a Subset 2

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Omit the first index to make a copy starting from the beginning of the container.

```
>>> tu[:2]
(23, 'abc')
```

- Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> tu[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Sequence

- To make a copy of an entire sequence, you can use [:].

```
>>> tu[:]
(23, 'abc', 4.56, (2,3), 'def')
```

- Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1       # 2 names refer to 1 reference
                        # Changing one affects both
>>> list2 = list1[:]    # Two independent copies, two refs
```

The ‘in’ Operator

- Boolean test whether a value is inside a container:

```
>>> li = [1, 2, 4, 5]
>>> 3 in li
False
>>> 4 in li
True
>>> 4 not in li
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- Be careful: the in keyword is also used in the syntax of for loops and list comprehensions.

The + Operator

- The + operator produces **a new string, tuple, or list** whose value is the concatenation of its arguments.

```
>>> "Hello" + " " + "World"
'Hello World'
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

The * Operator

- The * operator produces **a new string, tuple, or list** that “repeats” the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> "Hello" * 3
'HelloHelloHello'
```

Operations on Lists Only 1

```
>>> li = [1, 11, 3, 4, 5]
>>> li.append('a') # Our first exposure to method syntax
>>> li
[1, 11, 3, 4, 5, 'a']
>>> li.insert(2, 'i')
>>> li
[1, 11, 'i', 3, 4, 5, 'a']
```

The extend method vs the + operator.

- + creates a fresh list (with a new memory reference)
- extend operates on list li in place.

```
>>> li.extend([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

- Confusing:
  - Extend takes a list as an argument.
  - Append takes a singleton as an argument.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operations on Lists Only 3

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')        # index of first occurrence
1
>>> li.count('b')        # number of occurrences
2
>>> li.remove('b')       # remove first occurrence
>>> li
['a', 'c', 'b']
```

Operations on Lists Only 4

```
>>> li = [5, 2, 6, 8]
>>> li.reverse()  # reverse the list *in place*
>>> li
[8, 6, 2, 5]
>>> li.sort()      # sort the list *in place*
>>> li
[2, 5, 6, 8]
>>> li.sort(some_function) # sort in place using user-
defined comparison
```

Sets: A set type

- Sets store unordered, finite sets of unique, **immutable** objects
  - Sets are mutable, cannot be indexed.
  - Defined using { } (and commas) or `set()`
  - Common uses:
    - fast membership testing
    - removing duplicates from a sequence
    - computing mathematical operations such as intersection, union

```
>>> se1 = set()                       # create an empty set
>>> se2 = {"arrow", 1, 5.6}           # create another set
>>> se2.add("hello")
>>> print(se2)
{1, 'hello', 5.6, 'arrow'}
>>> se2.remove("hello")
>>> print(se2)
{1, 5.6, 'arrow'}
```

Dictionaries: A Mapping type

- Dictionaries store a mapping between a set of keys and a set of values.
  - Dictionaries are **mutable**
  - Keys can be any immutable type.
  - Values can be any type
  - A single dictionary can store values of different types
  - Defined using { } : (and commas).
- You can define, modify, view, lookup, and delete the key-value pairs in the dictionary.

Using 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

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

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

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> del d['user']           # Remove one.
>>> d
{'p':1234, 'i':34}
>>> d.clear()               # Remove all.
>>> d
{}

>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> d.keys()              # List of keys.
['user', 'p', 'i']
>>> d.values()            # List of values.
['bozo', 1234, 34]
>>> d.items()          # List of item tuples.
[('user','bozo'), ('p',1234), ('i',34)]
```

Control of flow 1

- The `if`/`elif`/`else` statement

```
if x == 3:
    print("x equals 3.")
elif x == 2:
    print("x equals 2.")
else:
    print("x equals something else.")
print("This is outside the if statement.")
```

Control of flow 2

- The **while** statement

```
x = 3
while x < 10:
   if x > 7:
        x += 2
        continue
   x = x + 1
   print("still in the loop.")
   if x == 8:
        break
print("Outside of the loop.")
```

Control of flow 3

- The `for` statement

```
for x in range(10):
   if x > 7:
        x += 2
        continue
   x = x + 1
   print("still in the loop.")
   if x == 8:
        break
print("Outside of the loop.")
```

range()

- The range() function has two sets of parameters, as follows:
  - range(stop)
    - stop: Number of integers (whole numbers) to generate, starting from zero. E.g. range(3) == [0, 1, 2].
  - range([start], stop[, step])
    - start: Starting number of the sequence.
    - stop: Generate numbers up to, but not including this number.
    - step: Difference between each number in the sequence.
- Note that:
  - All parameters must be integers.
  - All parameters can be positive or negative.

User-defined functions

- `def` creates a function and assigns it a name
- `return` sends a result back to the caller
- Arguments are passed by assignment
- Arguments and return types are not declared

```
def <name>(arg1, arg2, ..., argN):
    <statements>
    return <values>

def times(x,y):
     return x*y
```

Passing Arguments to Functions

- Arguments are passed by assignment
- Passed arguments are assigned to local names
- There is no call-by-reference per se since:
  - changes to immutable objects within a function only change what object the name points to (and **do not affect the caller**, unless it's a global variable)

- For immutable objects (e.g., integers, strings, tuples), Python acts like C's pass by value
- For mutable objects (e.g., lists), Python acts like C's pass by pointer; in-place changes to mutable objects can affect the caller

Example.py

```
def f1(x,y):
    x = x + 1
    y = y * 2
    print(x, y)      # 1 [1, 2, 1, 2]
def f2(x,y):
    x = x + 1
    y[0] = y[0] * 2
    print(x, y)      # 1 [2, 2]
a = 0                # immutable
b = [1,2]            # mutable
f1(a,b)
print(a, b)          # 0 [1, 2]
f2(a,b)
print(a, b)          # 0 [2, 2]
```

Optional Arguments

- Can define defaults for arguments that need not be passed

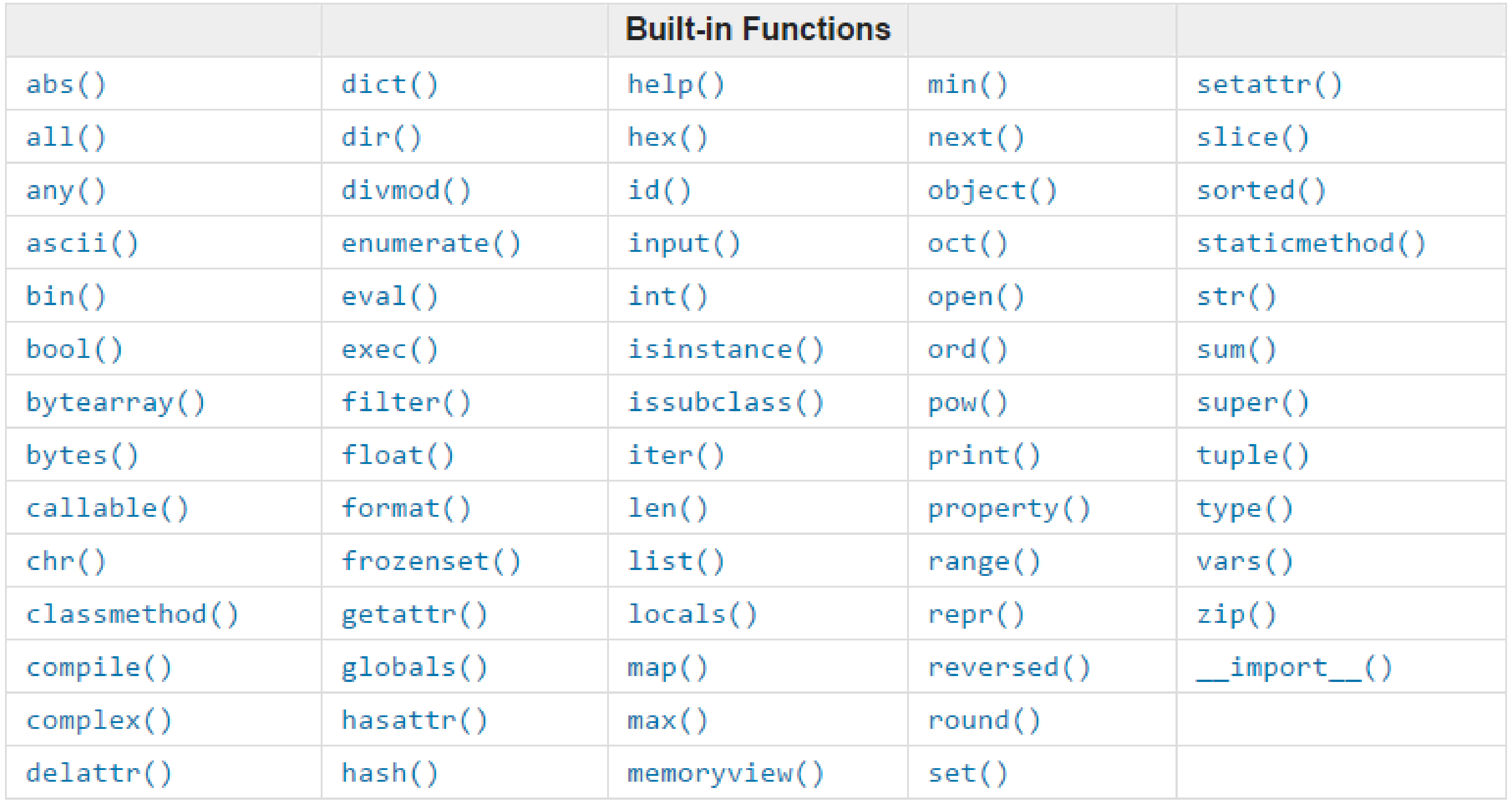
```
def func(a, b, c=10, d=100):
    print(a,b,c,d)

>>> func(1,2)
1 2 10 100
>>> func(1,2,3,4)
1 2 3 4
```

Important notes

- All functions in Python have a return value
  - even if no return line inside the code.
- Functions without a return, return the special value `None`.
- There is no function overloading in Python.
  - Two different functions can’t have the same name, even if they have different arguments.
- Functions can be used as any other data type. They can be:
  - Arguments to function
  - Return values of functions
  - Assigned to variables
  - Parts of tuples, lists, etc

Built-in functions

- https://docs.python.org/3/library/functions.html



		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

`len()` :

- Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

`min()` / `max()` :

- Return the smallest / largest item in an iterable or the smallest of two or more arguments.

Built-in functions: len(), max(), min()

```
>>> my_list = ['one', 'two', 3]
>>> my_list_len = len(my_list)
>>> for i in range(0, my_list_len):
...     print(my_list[i])
...
one
two
3
>>> max("hello","world")
'world'
>>> max(3,13)
13
>>> min([11,5,19,66])
5
```

Lambda

- Shorthand version of def statement; Useful for “inlining” functions
- Can only contain an expression in the function definition, not a block of statements (e.g., no if statements, etc)
- A `lambda` returns a function; the programmer can decide whether or not to assign this function to a name
- Simple example:

```
>>> def sum(x,y): return x+y
>>> ...
>>> sum(1,2)
3
>>> sum2 = lambda x, y: x+y
>>> sum2(1,2)
3
```

Built-in functions: map()

- map(func, seq) calls a given function on every element of a sequence and returns an iterator (not a list as in Python 2)
- map.py:

```
def double(x):
    return x*2
a = [1, 2, 3]
print(map(double, a)) # <map object at 0x000001B0512EDD30>
print(list(map(double, a)))             # [2, 4, 6]
```

- Alternatively (without def):

```
 a = [1, 2, 3]
 print(list(map((lambda x: x*2), a)))  # [2, 4, 6]
```

Built-in functions: map()

- map() can be applied to more than one sequence
- sequences have to have the same length
- map() will apply its lambda function to the elements of the argument sequences, i.e. it first applies to the elements with the 0th index, then to the elements with the 1st index until the n-th index is reached:

```
>>> a = [1,2,3,4]
>>> b = [17,12,11,10]
>>> c = [-1,-4,5,9]
>>> list(map(lambda x,y:x+y, a,b))
[18, 14, 14, 14]
>>> list(map(lambda x,y,z:x+y+z, a,b,c))
[17, 10, 19, 23]
>>> list(map(lambda x,y,z : 2.5*x + 2*y - z, a,b,c))
[37.5, 33.0, 24.5, 21.0]
```

Built-in functions: filter()

- filter(func, seq) filters out all the elements of a sequence for which the function returns True
  - Function has to return a Boolean value
- Example: filter out first the odd and then the even elements of the sequence of the first 11 Fibonacci numbers:

```
>>> fibonacci = [0,1,1,2,3,5,8,13,21,34,55]
>>> odd_numbers = list(filter(lambda x: x % 2, fibonacci))
>>> print(odd_numbers)
[1, 1, 3, 5, 13, 21, 55]
>>> even_numbers = list(filter(lambda x: x % 2 == 0, 
fibonacci))
>>> print(even_numbers)
[0, 2, 8, 34]
```

Modules

- Modules are functions and variables defined in separate files
- Items are imported using from or import

```
from module import function
function()
```

A’ Τρόπος

```
import module
module.function()
```

B’ Τρόπος

- Modules are namespaces
  - Can be used to organize variable names
    - i.e. atom.position = atom.position - molecule.position

Mathematical functions

- https://docs.python.org/3.6/library/math.html

```
>>> import math
>>> print(math.sqrt(3))
1.7320508075688772

>>> from math import sqrt
>>> print(sqrt(3))
1.7320508075688772
```

Objects

- A software item that contains **variables (attributes)** and **methods**
- Example:

```
class Student:
    """ A class representing a student """
    # constructor
    def __init__(self,n,a):
        # list the fields
        self.name = n
        self.age = a
    def getName(self):
        return self.age
```

Instantiating Objects

- There is no “new” keyword as in Java.
- Just use the class name with ( ) notation and assign the result to a variable
- `__init__` serves as a constructor for the class. Usually does some initialization work
- The arguments passed to the class name are given to its `__init__()` method
- So, the __init__ method for student is passed “Bob” and 21 and the new class instance is bound to b:

```
        b = student(“Bob”, 21)
```

Constructor: __init__

- An `__init__` method can take any number of arguments.
- Like other functions or methods, the arguments can be defined with default values, making them optional to the caller.
- However, the first argument `self` in the definition of __init__ is special…

self

- The first argument of every method is a reference to the current instance of the class
- By convention, we name this argument ***self***
- In `__init__`, *`self`* refers to object currently being created; in other class methods, it refers to the instance whose method was called
- Similar to the keyword *this* in Java or C++
  - But Python uses *self* more often than Java uses *this*
- Although you must specify *`self`* explicitly when *defining* the method, you don’t include it when *calling* the method.
  - Python passes it for you automatically

Defining a method:
(this code inside a class definition.)

```
def set_age(self, num):
        self.age = num
```

Calling a method:

```
>>> x.set_age(23)
```

Deleting instances: No Need to “free”

- When you are done with an object, you don’t have to delete or free it explicitly.
- Python has automatic **garbage collection**.
- Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
- Generally works well, few memory leaks
- There’s also no “destructor” method for classes

Access to Attributes and Methods

```
>>> f = student("Bob Smith", 23)

>>> f.full          # Access an attribute
"Bob"

>>> f.get_age()     # Access a method
23
```

Inheritance

- A class can *extend* the definition of another class
  - Allows use (or extension) of methods and attributes already defined in the previous one.
  - New class: *subclass*. Original: *parent*, *ancestor* or *superclass*
- To define a subclass, put the name of the superclass in parentheses after the subclass’s name on the first line of the definition.

```
class Cs_student(student):
```

  - Python has no ‘extends’ keyword like Java.
  - Multiple inheritance is supported.

Redefining Methods

- To *redefine a method* of the parent class, include a new definition using the same name in the subclass.
  - The old code won’t get executed.
- To execute the method in the parent class *in addition to* new code for some method, explicitly call the parent’s version of the method.

```
parentClass.methodName(self, a, b, c)
```

  - **The only time you ever explicitly pass ‘self’ as an argument is when calling a method of an ancestor.**

Definition of a class extending student

```
Class Student:
   “A class representing a student.”

   def __init__(self,n,a):
        self.full_name = n
        self.age = a

   def get_age(self):
        return self.age
```

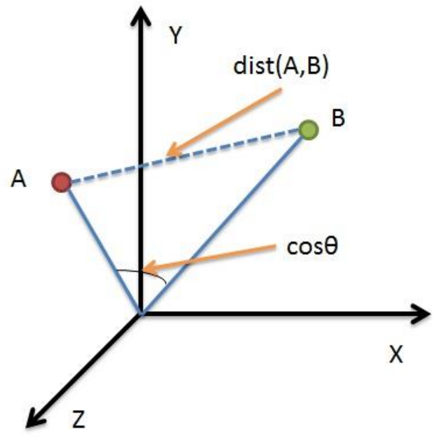
```
Class Cs_student (student):
   “A class extending student.”

   def __init__(self,n,a,s):
        student.__init__(self,n,a)     #Call __init__ for student
        self.section_num = s

   def get_age():                      #Redefines get_age method entirely
        print “Age: ” + str(self.age)
```

Let the game begin…

Implement 5 popular similarity/distance measures in python

Similarity and Distance

- Similarity is the measure of how alike two data objects are.
- Usually is related to distance with dimensions representing features of the objects
  - similarity is inversely proportional to distance



Features of each object (3D point):
A (x_1, y_1, z_1)
B (x_2, y_2, z_2)

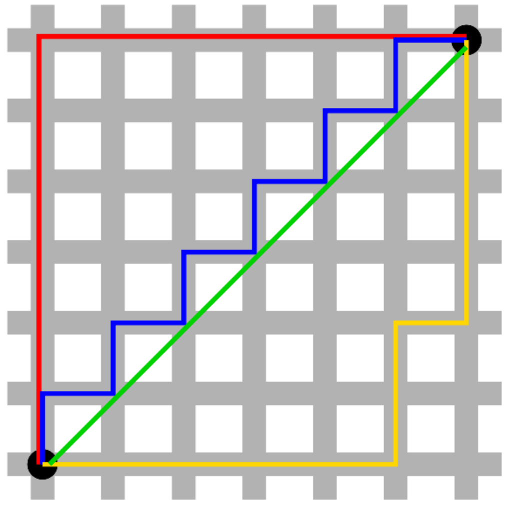
- Similarity is subjective and is highly dependent on the domain and application
  - For example, two fruits can be similar because of their color, size or taste
- How to calculate distance across dimensions/features that are unrelated?
  - relative values of each feature must be normalized

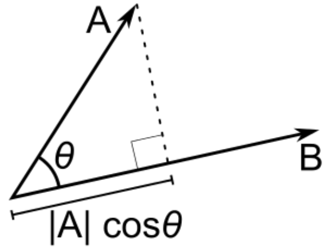
Similarity/Distance Measures

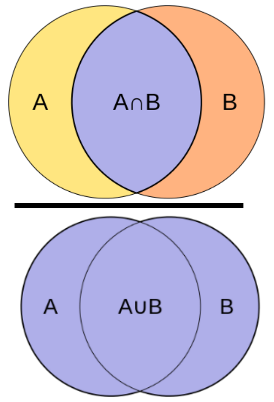
- Similarity is measured in the range 0 to 1 [0,1]. E.g. for 2 objects A, B:
  - Similarity = 1 if A = B
  - Similarity = 0 if A ≠ B
- Need for similarity measures
  - compare two lists of numbers (i.e. vectors), and compute a single number which evaluates their similarity
- Five (5) popular similarity/distance measures
  - Euclidean distance
  - Manhattan distance
  - Minkowski distance
  - Cosine similarity
  - Jaccard similarity

Similarity/Distance Measures

- $Euclidean\ Distance(A,B) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$
  - i is the number of features (or dimensions or variables)
- $Manhattan\ Distance(A,B) = \sum_{i=1}^{n}|a_i - b_i|$
- $Minkowski\ Distance(A,B) = \sqrt[\lambda]{\sum_{i=1}^{n}(a_i - b_i)^{\lambda}}$
  - λ=1 : Manhattan distance – λ=2 : Euclidean distance
- $Cosine\ Similarity(A,B) = cos\theta = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} a_i * b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} * \sqrt{\sum_{i=1}^{n} b_i^2}}$
- $Jaccard\ Similarity(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} =$
  - $JaccardDistance = 1 - Jaccard(A,B)$







Exercise

- Dataset: a .csv file with energy consumption data for one house from January 2015 to January 2016
  - Each line contains a <timestamp, energy> pair

```
timestamp,energy
01-01-15 19:30,350
01-01-15 19:45,204
01-01-15 20:00,225
```

  - Energy value: electricity consumption (Wh) for the previous 15-minute slot
- Evaluate similarity (distance) in consumption between two days
  - Create 2 lists; each list to contain 96 values (00:00 – 23:45) for one day
- Download Lab4.py and Lab4.csv files
  - Source code for creating 2 lists is ready
- Implement 5 functions to evaluate the similarity (distance)