matrixBi

Python for AI Engineer

Oran Shemesh

matrixBi

About me

- More than 10 years of experience in Programing in professional roles in multinational organizations and start-ups
- Big-data expert, with expertise in Data Scientist, Data Engineer, Spark Developer and Dev-ops
- Currently working as a CTO at ANS Tech

matrixBi

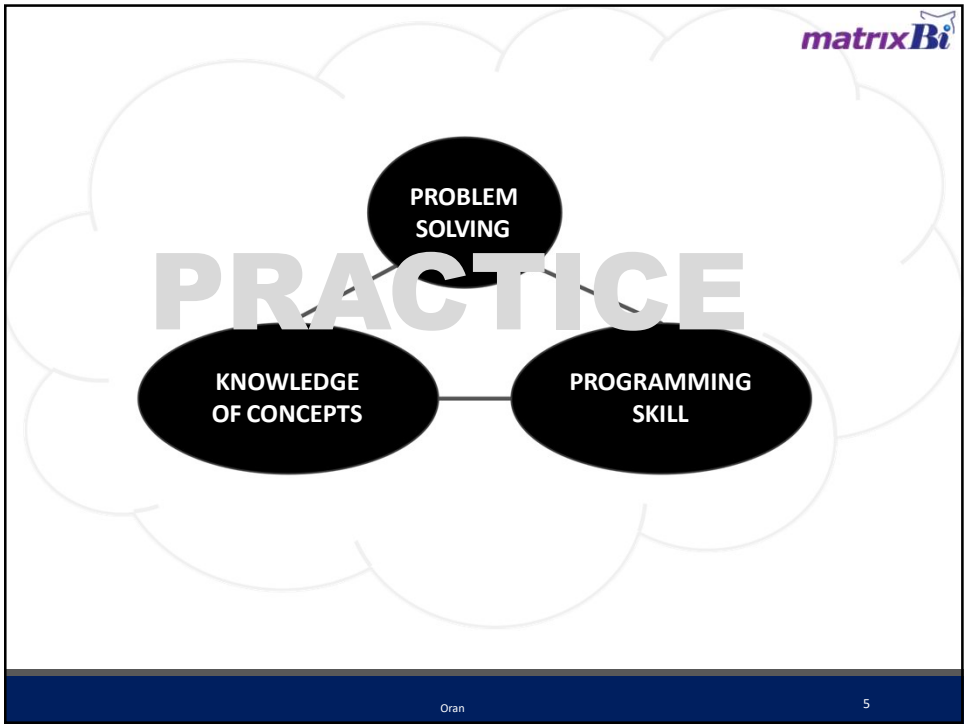
TODAY

- Course info
- Material learned at home:
  - Python basics
  - Conditions, Loops, Functions
  - Object Oriented Programming
- Investigate and present data:
  - Numpy
  - MatplotLib

matrixBi

FAST PACED COURSE

- Position yourself to succeed!
  - It's is not about the number of hours you practice, it's about the number of hours your mind is present during the practice." – Kobe Bryant.
  - **write notes** and come back to them later
- **PRACTICE. PRACTICE? PRACTICE!**
  - can't passively absorb programming as a skill
  - download code before lecture and follow along
  - do exercises
  - **don't be afraid to try out Python commands!**



matrixBi

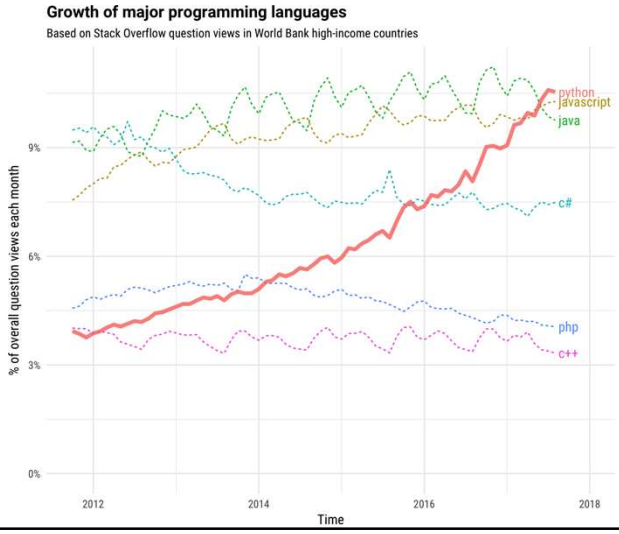
COURSE POLICIES

- Collaboration
  - must collaborate with anyone
  - required to write code independently and write names of all collaborators on submission
  - I will be running the code solution with you
- Unlike what we were taught in school
  - learning is the acquisition of the ability to apply things
- An unfamiliar word
  - "עם רדת נשף, הילדים היו עצובים יותר ובהעדרו, הם היו שמחים הרבה יותר"

matrixBi

Growth major languages

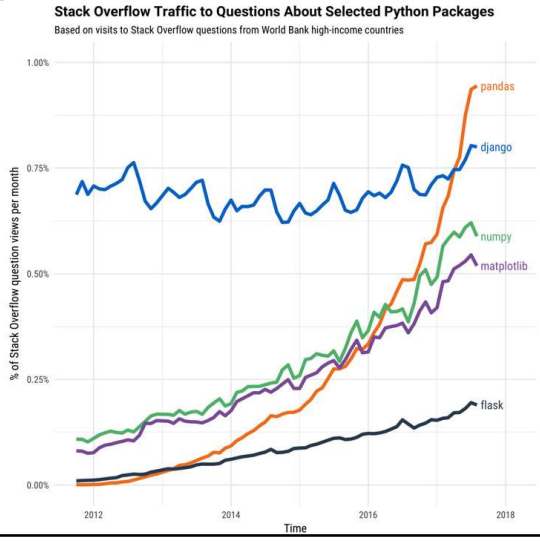
link



matrixBi

Stack overflow traffic to questions about selected python packages

link



matrixBi

SCALAR OBJECTS

- `int` – represent **integers**, ex. `5`
- `float` – represent **real numbers**, ex. `3.27`
- `bool` – represent **Boolean** values `True` and `False`
- `NoneType` – **special** and has one value, `None`
- can use `type()` to see the type of an object

```
>>> type(5)
int
>>> type(3.0)
float
```

what you write into the Python shell

what shows after hitting enter

matrixBi

TYPE CONVERSIONS (CAST)

- can **convert object of one type to another**
- `float(3)` converts integer `3` to float `3.0`
- `int(3.9)` truncates float `3.9` to integer `3`

matrixBi

PRINTING TO CONSOLE

- to show output from code to a user, use `print` command

```
In [11]: 3+2
Out[11]: 5

In [12]: print(3+2)
5
```

"Out" tells you it's an interaction within the shell only

No "Out" means it is actually shown to a user, apparent when you edit/run files

matrixBi

STRINGS

- letters, special characters, spaces, digits
- enclose in **quotation marks or single quotes**

```
hi = "hello there"
```

- **concatenate** strings

```
name = "Oran"

greet = hi + name

greeting = hi + " " + name
```

- do some **operations** on a string as defined in Python docs

```
silly = hi + " " + name * 3
```

matrixBi

INPUT/OUTPUT: `print`

- used to **output** stuff to console
- keyword is `print`

```
x = 1
print(x)
x_str = str(x)
print("my fav num is", x, ".", "x =", x)
print("my fav num is " + x_str + ". " + "x = " + x_str)
print("my fav num is {}. {}".format(x_str, x))
print(f"my fav num is {x_str}. {x=}")
```

matrixBi

INPUT/OUTPUT: `input("")`

- prints whatever is in the quotes
- user types in something and hits enter
- binds that value to a variable

```
text = input("Type anything... ")
print(5*text)
```

- `input` **gives you a string** so must cast if working with numbers

```
num = int(input("Type a number... "))
print(5*num)
```

matrixBi

COMPARISON OPERATORS ON `int, float, string`

- `i` and `j` are variable names
- comparisons below evaluate to a Boolean

`i > j`

`i >= j`

`i < j`

`i <= j`

`i == j` → **equality** test, `True` if `i` is the same as `j`

`i != j` → **inequality** test, `True` if `i` not the same as `j`

matrixBi

LOGIC OPERATORS ON bools

- `a` and `b` are variable names (with Boolean values)

`not a` → `True` if `a` is `False`
`False` if `a` is `True`

`a and b` → `True` if both are `True`

`a or b` → `True` if either or both are `True`

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

matrixBi

CONTROL FLOW - BRANCHING

```
if <condition>:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
elif <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

- `<condition>` has a value `True` or `False`
- evaluate expressions in that block if `<condition>` is `True`

CONTROL FLOW: `while` LOOPS

matrixBi

```
while <condition>:
    <expression>
    <expression>
    ...
```

- `<condition>` evaluates to a Boolean
- if `<condition>` is `True`, do all the steps inside the while code block
- check `<condition>` again
- repeat until `<condition>` is `False`

matrixBi

`while` LOOP EXAMPLE

```
You are in the Lost Forest.
************
************
  ☺
************
************
Go left or right?
```

PROGRAM:

```
n = input("You're in the Lost Forest. Go left or right? ")
while n == "right":
    n = input("You're in the Lost Forest. Go left or right? ")
print("You got out of the Lost Forest!")
```

matrixBi

CONTROL FLOW: `while` and `for` LOOPS

- iterate through numbers in a sequence

```
# more complicated with while loop
n = 0
while n < 5:
    print(n)
    n = n+1


# shortcut with for loop
for n in range(5):
    print(n)
```

matrixBi

CONTROL FLOW: `for` LOOPS

```
for <variable> in range(<some_num>):
    <expression>
    <expression>
    ...
```

- each time through the loop, `<variable>` takes a value
- first time, `<variable>` starts at the smallest value
- next time, `<variable>` gets the prev value + 1
- etc.

matrixBi

`range(start,stop,step)`

- default values are `start = 0` and `step = 1` and optional
- loop until value is `stop - 1`

```
mysum = 0
for i in range(7, 10):
    mysum += i
print(mysum)
```

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
print(mysum)
```

matrixBi

`break` STATEMENT

- immediately exits whatever loop it is in
- skips remaining expressions in code block
- exits only innermost loop!

```
while <condition_1>:
    while <condition_2>:
        <expression_a>
        break
        <expression_b>
    <expression_c>
```

matrixBi

`break` STATEMENT

```
mysum = 0
for i in range(5, 11, 2):
    if mysum == 5:
        break
    mysum += 1
print(mysum)
```

- what happens in this program?

matrixBi

`for` VS `while` LOOPS

`for` loops

- **know** number of iterations
- can **end early** via `break`
- uses a **counter**
- **can rewrite** a `for` loop using a `while` loop

`while` loops

- **unbounded** number of iterations
- can **end early** via `break`
- can use a **counter but must initialize** before loop and increment it inside loop
- **may not be able to rewrite** a `while` loop using a `for` loop

matrixBi

STRINGS

- think of as a **sequence** of case sensitive characters
- can compare strings with ==, >, < etc.
- `len()` is a function used to retrieve the **length** of the string in the parentheses

```
s = "abc"
len(s)
```
→ evaluates to 3

matrixBi

STRINGS

- square brackets used to perform **indexing** into a string to get the value at a certain index/position

```
s = "abc"
```

index: 0 1 2 ← indexing always starts at 0

index: -3 -2 -1 ← last element always at index -1

```
s[0]   →
s[1]   →
s[2]   →
s[3]   →
s[-1]  →
s[-2]  →
s[-3]  →
```

matrixBi

STRINGS

- can **slice** strings using `[start:stop:step]`
- if give two numbers, `[start:stop]`, `step=1` by default
- you can also omit numbers and leave just colons

If unsure what some command does, try it out in your console!

```
s = "abcdefgh"
s[3:6]   →
s[3:6:2] →
s[::]    →
s[::-1]  →
s[4:1:-2]→
```

matrixBi

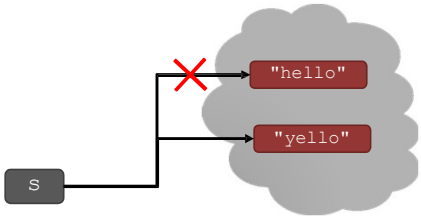
STRINGS

- strings are "**immutable**" – cannot be modified

```
s = "hello"
```

```
s[0] = 'y'
```
→ gives an error

```
s = 'y'+s[1:len(s)]
```
→ is allowed, s bound to new object



matrixBi

STRINGS AND LOOPS

- these two code snippets do the same thing
- bottom one is more "pythonic"

```
s = "abcdefgh"
for index in range(len(s)):
    if s[index] == 'i' or s[index] == 'u':
        print("There is an i or u")

for char in s:
    if char == 'i' or char == 'u':
        print("There is an i or u")
```

matrixBi

TUPLES

- an ordered sequence of elements, can mix element types
- cannot change element values, **immutable**
- represented with parentheses

remember strings?

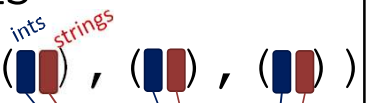
```
te = ()
t = (2,"mit",3)
t[0]                    → evaluates to 2
(2,"mit",3) + (5,6)     → evaluates to (2,"mit",3,5,6)
t[1:2]                  → slice tuple, evaluates to ("mit",)
t[1:3]                  → slice tuple, evaluates to ("mit",3)
len(t)                  → evaluates to 3
t[1] = 4                → gives error, can't modify object
```

empty tuple

extra comma means a tuple with one element

matrixBi

MANIPULATING TUPLES



- can **iterate** over tuples

```
def get_data(aTuple):
    nums = ()
    words = ()
    for t in aTuple:
        nums = nums + (t[0],)
        if t[1] not in words:
            words = words + (t[1],)
    min_n = min(nums)
    max_n = max(nums)
    unique_words = len(words)
    return (min_n, max_n, unique_words)
```

empty tuple

singleton tuple

nums( )

words(? ? ?)

if not already in words
i.e. unique strings from aTuple

matrixBi

LISTS

- **ordered sequence** of information, accessible by index
- a list is denoted by **square brackets**, `[]`
- a list contains **elements**
  - usually homogeneous (ie, all integers)
  - can contain mixed types (not common)
- list elements can be changed so a list is **mutable**

matrixBi

INDICES AND ORDERING

```
a_list = []
```
empty list

```
L = [2, 'a', 4, [1,2]]
len(L)  →
L[0]    →
L[2]+1  →
L[3]    →
L[4]    →
i = 2
L[i-1]  →
```

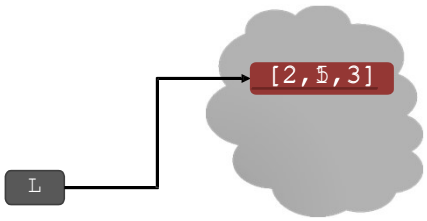
matrixBi

CHANGING ELEMENTS

- lists are **mutable**!
- assigning to an element at an index changes the value

```
L = [2, 1, 3]
L[1] = 5
```

- `L` is now `[2, 5, 3]`, note this is the **same object** `L`



matrixBi

ITERATING OVER A LIST

- compute the **sum of elements** of a list
- common pattern, iterate over list elements

like strings, can iterate over list elements directly

```
total = 0
for i in range(len(L)):
    total += L[i]
print total
```

```
total = 0
for i in L:
    total += i
print total
```

- notice
  - list elements are indexed `0` to `len(L)-1`
  - `range(n)` goes from `0` to `n-1`

matrixBi

CONVERT LISTS TO STRINGS AND BACK

- convert **string to list** with `list(s)`, returns a list with every character from `s` an element in `L`
- can use `s.split()`, to **split a string on a character** parameter, splits on spaces if called without a parameter

matrixBi

MUTATION, ALIASING, CLONING

Again, Python Tutor is your best friend to help sort this out!

http://www.pythontutor.com/

matrixBi

LISTS IN MEMORY

- lists are **mutable**
- behave differently than immutable types
- is an object in memory
- variable name points to object
- any variable pointing to that object is affected
- key phrase to keep in mind when working with lists is **side effects**

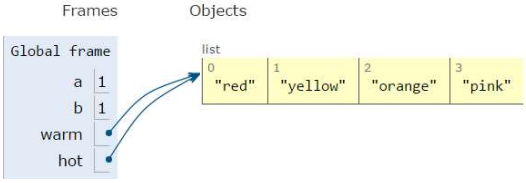
matrixBi

ALIASES

- `hot` is an **alias** for `warm` – changing one changes the other!
- `append()` has a side effect

```
1  a = 1
2  b = a
3  print(a)
4  print(b)
5
6  warm = ['red', 'yellow', 'orange']
7  hot = warm
8  hot.append('pink')
9  print(hot)
10 print(warm)
```

```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```



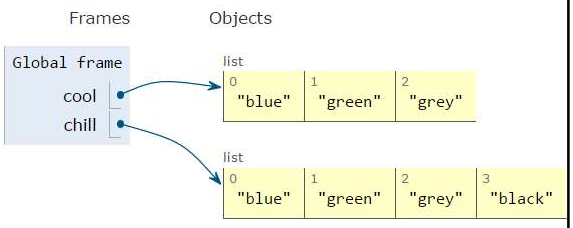
matrixBi

CLONING A LIST

- create a new list and **copy every element** using `chill = cool[:]`

```
1 cool = ['blue', 'green', 'grey']
2 chill = cool[:]
3 chill.append('black')
4 print(chill)
5 print(cool)
```

```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```



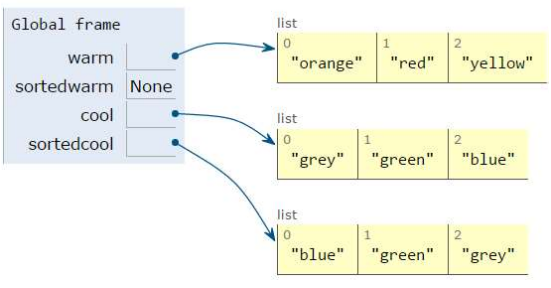
matrixBi

SORTING LISTS

- calling `sort()` **mutates** the list, returns nothing
- calling `sorted()` **does not mutate** list, must assign result to a variable

```
1 warm = ['red', 'yellow', 'orange']
2 sortedwarm = warm.sort()
3 print(warm)
4 print(sortedwarm)
5
6 cool = ['grey', 'green', 'blue']
7 sortedcool = sorted(cool)
8 print(cool)
9 print(sortedcool)
```

```
['orange', 'red', 'yellow']
None
['grey', 'green', 'blue']
['blue', 'green', 'grey']
```



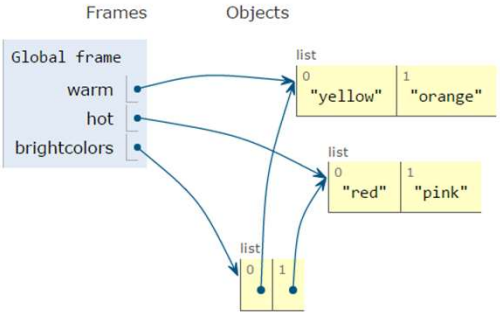
matrixBi

LISTS OF LISTS OF LISTS OF….

- can have **nested** lists
- side effects still possible after mutation

```
[['yellow', 'orange'], ['red']]
['red', 'pink']
[['yellow', 'orange'], ['red', 'pink']]
```

```
1 warm = ['yellow', 'orange']
2 hot = ['red']
3 brightcolors = [warm]
4 brightcolors.append(hot)
5 print(brightcolors)
6 hot.append('pink')
7 print(hot)
8 print(brightcolors)
```



matrixBi

MUTATION AND ITERATION
Try this in Python Tutor!

- **avoid** mutating a list as you are iterating over it

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)
```

✖

```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

✓

```
L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups(L1, L2)
```

clone list first, note that `L1_copy = L1` does NOT clone

- `L1` is `[2,3,4]` not `[3,4]` Why?
  - Python uses an internal counter to keep track of index it is in the loop
  - mutating changes the list length but Python doesn't update the counter
  - loop never sees element 2

matrixBi

HOW DO WE WRITE CODE?

- so far…
  - covered language mechanisms
  - know how to write different files for each computation
  - each file is some piece of code
  - each code is a sequence of instructions
- problems with this approach
  - easy for small-scale problems
  - messy for larger problems
  - hard to keep track of details
  - how do you know the right info is supplied to the right part of code

matrixBi

GOOD PROGRAMMING

- more code not necessarily a good thing
- measure good programmers by the amount of functionality
- introduce **functions**
- mechanism to achieve **decomposition** and **abstraction**

matrixBi

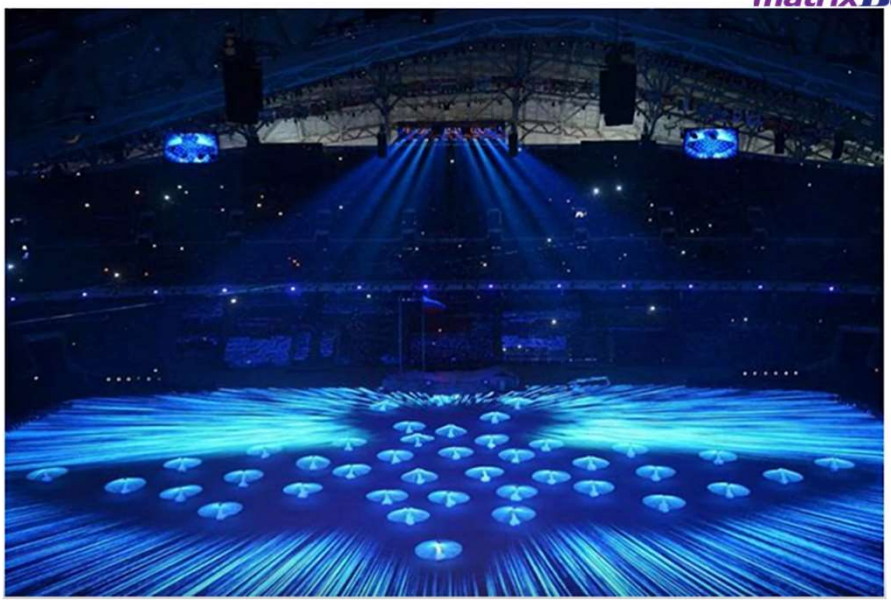
EXAMPLE – PROJECTOR

matrixBi

EXAMPLE – PROJECTOR

- a projector is a black box
- don't know how it works
- know the interface: input/output
- connect any electronic to it that can communicate with that input
- black box somehow converts image from input source to a wall, magnifying it
- **ABSTRACTION IDEA**: do not need to know how projector works to use it

matrixBi



matrixBi

EXAMPLE – PROJECTOR

- projecting large image for Olympics decomposed into separate tasks for separate projectors
- each projector takes input and produces separate output
- all projectors work together to produce larger image
- **DECOMPOSITION IDEA**: different devices work together to achieve an end goal

matrixBi

CREATE STRUCTURE with DECOMPOSITION

- in projector example, separate devices
- in programming, divide code into **modules**
  - are **self-contained**
  - used to **break up** code
  - intended to be **reusable**
  - keep code **organized**
  - **keep code coherent**
- this lecture, achieve decomposition with **functions**
- in a few weeks, achieve decomposition with **classes**

matrixBi

SUPRESS DETAILS with ABSTRACTION

- in projector example, instructions for how to use it are sufficient, no need to know how to build one
- in programming, think of a piece of code as a **black box**
  - cannot see details
  - do not need to see details
  - do not want to see details
  - hide tedious coding details
- achieve abstraction with **function specifications** or **docstrings**

matrixBi

FUNCTIONS

- write reusable pieces/chunks of code, called **functions**
- functions are not run in a program until they are "**called**" or "**invoked**" in a program
- function characteristics:
  - has a **name**
  - has **parameters** (0 or more)
  - has a **docstring** (optional but recommended)
  - has a **body**
  - **returns** something

matrixBi

HOW TO WRITE and CALL/INVOKE A FUNCTION

keyword

name

parameters or arguments

specification, docstring

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

body

later in the code, you call the function using its name and values for parameters

```
is_even(3)
```

matrixBi

IN THE FUNCTION BODY

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

run some commands

keyword

expression to evaluate and return

matrixBi

VARIABLE SCOPE

- **formal parameter** gets bound to the value of **actual parameter** when function is called
- new **scope/frame/environment** created when enter a function
- **scope** is mapping of names to objects

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x
```

formal parameter

Function definition

```
x = 3
z = f( x )
```

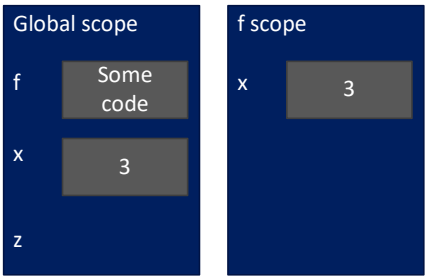
actual parameter

Main program code
* initializes a variable x
* makes a function call f(x)
* assigns return of function to variable z

matrixBi

VARIABLE SCOPE

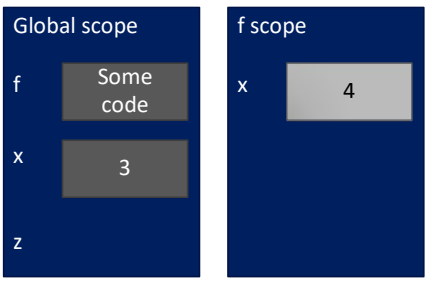
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



matrixBi

VARIABLE SCOPE

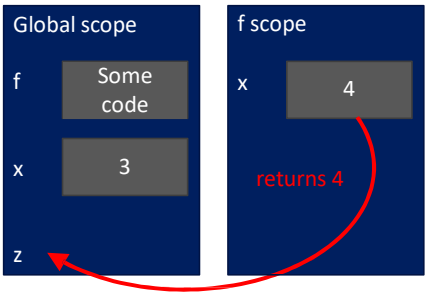
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



matrixBi

VARIABLE SCOPE

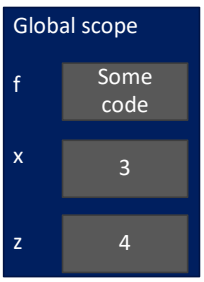
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



matrixBi

VARIABLE SCOPE

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



matrixBi

ONE WARNING IF NO `return` STATEMENT

```
def is_even( i ):
    """
    Input: i, a positive int
    Does not return anything
    """
    i%2 == 0
```

without a return statement

- Python returns the value **None, if no `return` given**
- represents the absence of a value

matrixBi

ONE WARNING IF NO `return` STATEMENT

```
def is_even( i ):
    """
    Input: i, a positive int
    Does not return anything
    """
    remainder = i % 2
is_even( 3 )
print(is_even( 3 ))
```

None

matrixBi

FUNCTIONS AS ARGUMENTS

- arguments can take on any type, even functions

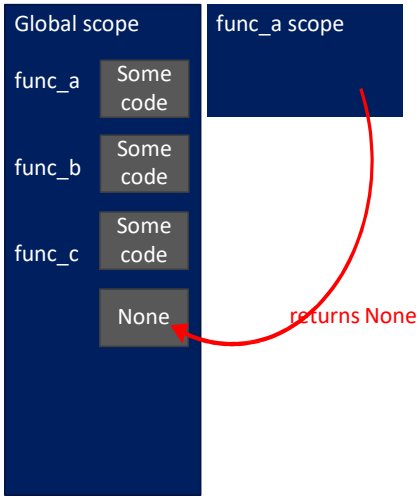
```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```

call `func_a`, takes no parameters
call `func_b`, takes one parameter
call `func_c`, takes one parameter, another function

matrixBi

FUNCTIONS AS ARGUMENTS

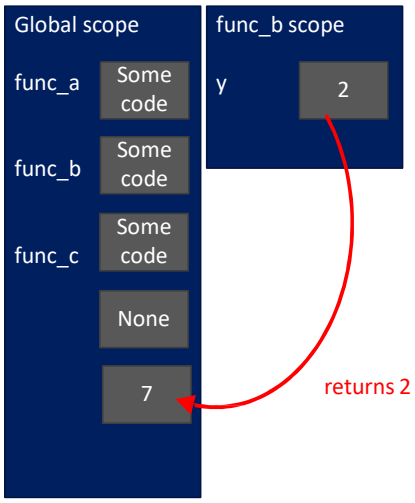
```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



matrixBi

FUNCTIONS AS ARGUMENTS

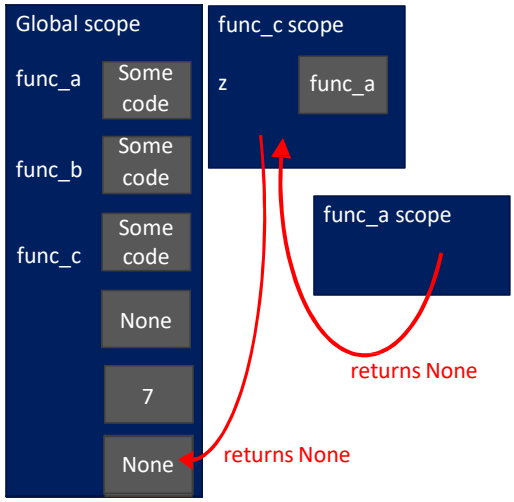
```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



matrixBi

FUNCTIONS AS ARGUMENTS

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



SCOPE EXAMPLE

- inside a function, **can access** a variable defined outside
- inside a function, **cannot modify** a variable defined outside -- can using **global variables**, but frowned upon

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

x = 5
f(x)
print(x)
```

x is re-defined in scope of f

different x objects

```
def g(y):
    print(x)
    print(x + 1)

x = 5
g(x)
print(x)
```

x from outside g

x inside g is picked up from scope that called function g

```
def h(y):
    x += 1

x = 5
h(x)
print(x)
```

UnboundLocalError: local variable 'x' referenced before assignment

SCOPE EXAMPLE

- inside a function, **can access** a variable defined outside
- inside a function, **cannot modify** a variable defined outside -- can using **global variables**, but frowned upon

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

x = 5
f(x)
print(x)
```

```
def g(y):
    print(x)


x = 5
g(x)
print(x)
```

```
def h(y):
    x += 1

x = 5
h(x)
print(x)
```

x from global/main program scope

matrixBi

HARDER SCOPE EXAMPLE

IMPORTANT and TRICKY!

Python Tutor is your best friend to help sort this out!

http://www.pythontutor.com/

matrixBi

LEGB Rule:

- L: Local — Names assigned in any way within a function (def or lambda), and not declared global in that function.
- E: Enclosing function locals — Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.
- G: Global (module) — Names assigned at the top-level of a module file, or declared global in a def within the file.
- B: Built-in (Python) — Names preassigned in the built-in names module : open, range, SyntaxError,...

matrixBi

```
# GLOBAL
name = 'THIS IS A GLOBAL STRING'

def greet():

    # ENCLOSING
    name = 'Sammy'

    def hello():
        #LOCAL
        name = 'IM A LOCAL'
        print('Hello '+name)

    hello()
```

matrixBi

FUNCTIONS AS ARGUMENTS

- arguments can take on any type, even functions

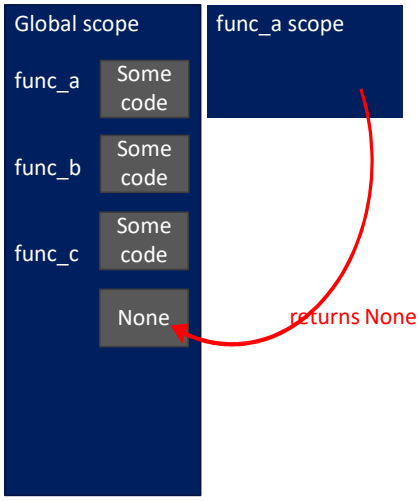
```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```

call `func_a`, takes no parameters
call `func_b`, takes one parameter
call `func_c`, takes one parameter, another function

matrixBi

FUNCTIONS AS ARGUMENTS

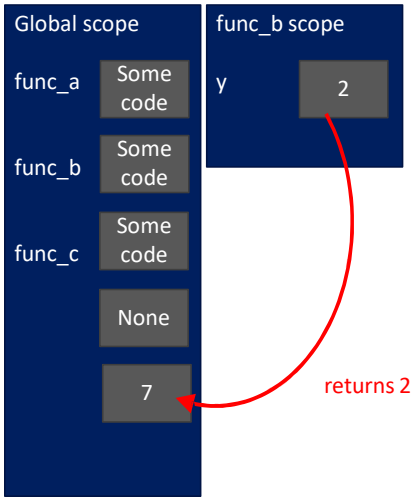
```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



matrixBi

FUNCTIONS AS ARGUMENTS

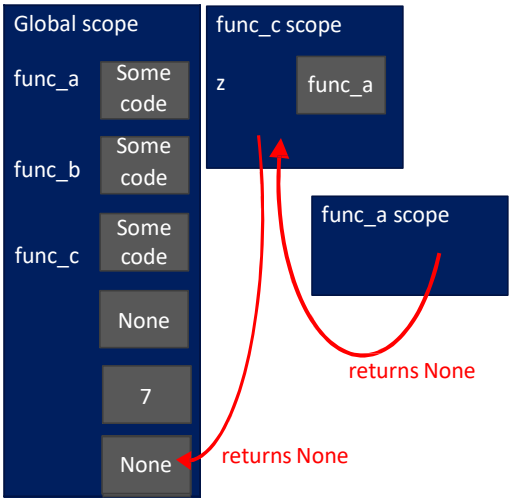
```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



matrixBi

FUNCTIONS AS ARGUMENTS

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



matrixBi

HOW TO STORE STUDENT INFO

- so far, can store using separate lists for every info

```
names = ['Ana', 'John', 'Denise', 'Katy']
grade = ['B', 'A+', 'A', 'A']
course = [2.00, 6.0001, 20.002, 9.01]
```

- a **separate list** for each item
- each list must have the **same length**
- info stored across lists at **same index**, each index refers to info for a different person

matrixBi

HOW TO UPDATE/RETRIEVE STUDENT INFO

```
def get_grade(student, name_list, grade_list, course_list):
    i = name_list.index(student)
    grade = grade_list[i]
    course = course_list[i]
    return (course, grade)
```

- **messy** if have a lot of different info to keep track of
- must maintain **many lists** and pass them as arguments
- must **always index** using integers
- must remember to change multiple lists

matrixBi

A BETTER AND CLEANER WAY – A DICTIONARY

- nice to **index item of interest directly** (not always int)
- nice to use **one data structure**, no separate lists

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
…	…

index element

A dictionary

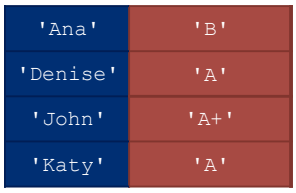
Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
…	…

custom index by label element

matrixBi

A PYTHON DICTIONARY

- store pairs of data
  - key
  - value



empty dictionary

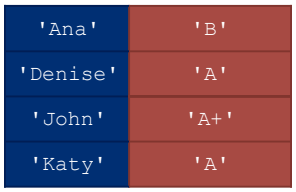
```
my_dict = {}
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

key1 val1 key2 val2 key3 val3 key4 val4

matrixBi

DICTIONARY LOOKUP

- similar to indexing into a list
- **looks up** the **key**
- **returns** the **value** associated with the key
- if key isn't found, get an error




```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
grades['John']      → evaluates to 'A+'
grades['Sylvan']    → gives a KeyError
```

matrixBi

DICTIONARY OPERATIONS

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'
'Sylvan'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- **add** an entry

```
grades['Sylvan'] = 'A'
```

- **test** if key in dictionary

```
'John' in grades          → returns True
'Daniel' in grades        → returns False
```

- **delete** entry

```
del(grades['Ana'])
```

matrixBi

DICTIONARY OPERATIONS

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- get an **iterable that acts like a tuple of all keys** (no guaranteed order)

```
grades.keys()   → returns ['Denise','Katy','John','Ana']
```

- get an **iterable that acts like a tuple of all values**

```
grades.values() → returns ['A', 'A', 'A+', 'B']
```

(no guaranteed order)

matrixBi

DICTIONARY KEYS and VALUES

- values
  - any type (**immutable and mutable**)
  - can be **duplicates**
  - dictionary values can be lists, even other dictionaries!
- keys
  - must be **unique**
  - **immutable** type (`int`, `float`, `string`, `tuple`, `bool`)
    - actually need an object that is **hashable,** but think of as immutable as all immutable types are hashable
  - careful with `float` type as a key
- **no order** to keys or values!

```
d = {4:{1:0}, (1,3):"twelve",
'const':[3.14,2.7,8.44]}
```

matrixBi

List vs dict

- **ordered** sequence of elements
- look up elements by an integer index
- indices have an **order**
- index is an **integer**

- **matches** "keys" to "values"
- look up one item by another item
- **no order** is guaranteed
- key can be any **immutable** type

matrixBi

OBJECTS

- Python supports many different kinds of data

```
1234      3.14159     "Hello"     [1, 5, 7, 11, 13]
{"CA": "California", "MA": "Massachusetts"}
```

- each is an **object**, and every object has:
  - a **type**
  - an internal **data representation** (primitive or composite)
  - a set of procedures for **interaction** with the object
- an object is an **instance** of a type
  - `1234` is an instance of an `int`
  - `"hello"` is an instance of a string

matrixBi

OBJECT ORIENTED PROGRAMMING (OOP)

- **EVERYTHING IN PYTHON IS AN OBJECT** (and has a type)
- can **create new objects** of some type
- can **manipulate objects**
- can **destroy objects**
  - explicitly using `del` or just "forget" about them
  - python system will reclaim destroyed or inaccessible objects – called "garbage collection"

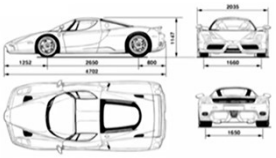
matrixBi

WHAT ARE OBJECTS?

- objects are **a data abstraction** that captures…

(1) an **internal representation**
- through data attributes

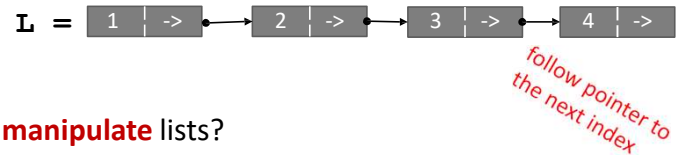
(2) an **interface** for interacting with object
- through methods (aka procedures/functions)
- defines behaviors but hides implementation



matrixBi

EXAMPLE: [1,2,3,4] has type list

- how are lists **represented internally**? linked list of cells



- how to **manipulate** lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- internal representation should be private
- correct behavior may be compromised if you manipulate internal representation directly

matrixBi

ADVANTAGES OF OOP

- **bundle data into packages** together with procedures that work on them through well-defined interfaces
- **divide-and-conquer** development
  - implement and test behavior of each class separately
  - increased modularity reduces complexity
- classes make it easy to **reuse** code
  - many Python modules define new classes
  - each class has a separate environment (no collision on function names)
  - inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

Implementing the class | Using the class

matrixBi

CREATING AND USING YOUR OWN TYPES WITH CLASSES

- make a distinction between **creating a class** and **using an instance** of the class
- **creating** the class involves
  - defining the class name
  - defining class attributes
  - *for example, someone wrote code to implement a list class*
- **using** the class involves
  - creating new **instances** of objects
  - doing operations on the instances
  - *for example,* `L=[1,2]` *and* `len(L)`

matrixBi

DEFINE YOUR OWN TYPES

- use the `class` keyword to define a new type

name/type

class parent

```
class Coordinate(object):
    #define attributes here
```

class definition

- similar to `def`, indent code to indicate which statements are part of the **class definition**
- the word `object` means that `Coordinate` is a Python object and **inherits** all its attributes (inheritance next lecture)
  - `Coordinate` is a subclass of `object`
  - `object` is a superclass of `Coordinate`

matrixBi

WHAT ARE ATTRIBUTES?

- data and procedures that "**belong**" to the class
- **data attributes**
  - think of data as other objects that make up the class
  - *for example, a coordinate is made up of two numbers*
- **methods** (procedural attributes)
  - think of methods as functions that only work with this class
  - how to interact with the object
  - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

Implementing the class | Using the class

matrixBi

DEFINING HOW TO CREATE AN INSTANCE OF A CLASS

- first have to define **how to create an instance** of object
- use a **special method called `__init__`** to initialize some data attributes

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

special method to create an instance — is double underscore

parameter to refer to an instance of the class

what data initializes a Coordinate object

two data attributes for every Coordinate

Implementing the class | Using the class

matrixBi

ACTUALLY CREATING AN INSTANCE OF A CLASS

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

create a new object of type Coordinate and pass in 3 and 4 to the __init__

use the dot to access an attribute of instance c

- data attributes of an instance are called **instance variables**

matrixBi

WHAT IS A METHOD?

- procedural attribute, like a **function that works only with this class**
- Python sometime passes the object as the first argument
  - convention is to use **`self`** as the name of the first argument of all methods
- the "**.**" **operator** is used to access any attribute
  - a data attribute of an object
  - a method of an object

Implementing the class | Using the class

matrixBi

DEFINE A METHOD FOR THE `Coordinate` CLASS

```
class Coordinate(object):
    def __init_(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

another parameter to method

use it to refer to any instance

dot notation to access data

- other than `self` and dot notation, methods behave just like functions (take params, do operations, return)

matrixBi

HOW TO USE A METHOD

```
def distance(self, other):
    # code here
```

method def

Using the class:

- conventional way

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

object to call method on; name of method; parameters not including self (self is implied to be c)

- equivalent to

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(Coordinate.distance(c, zero))
```

name of class; name of method; parameters, including an object to call the method on, representing self

matrixBi

PRINT REPRESENTATION OF AN OBJECT

```
>>> c = Coordinate(3,4)
>>> print(c)
<__main__.Coordinate object at 0x7fa918510488>
```

- **uninformative** print representation by default
- define a **__str__ method** for a class
- Python calls the `__str__` method when used with `print` on your class object
- you choose what it does! Say that when we print a `Coordinate` object, want to show

```
>>> print(c)
<3,4>
```

Implementing the class | Using the class

matrixBi

DEFINING YOUR OWN PRINT METHOD

```
class Coordinate(object):
    def __init_(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

name of special method

must return a string

Implementing the class | Using the class

matrixBi

WRAPPING YOUR HEAD AROUND TYPES AND CLASSES

- can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
>>> print(c)
<3,4>
>>> print(type(c))
<class __main_.Coordinate>
```

return of the __str__ method

the type of object c is a class Coordinate

- this makes sense since

```
>>> print(Coordinate)
<class __main__.Coordinate>
>>> print(type(Coordinate))
<type 'type'>
```

a Coordinate is a class

a Coordinate class is a type of object

- use `isinstance()` to check if an object is a `Coordinate`

```
>>> print(isinstance(c, Coordinate))
True
```

matrixBi

SPECIAL OPERATORS

- +, -, ==, <, >, len(), print, and many others

https://docs.python.org/3/reference/datamodel.html#basic-customization

- like `print`, can override these to work with your class
- define them with double underscores before/after

```
__add__(self, other)     →    self + other
__sub__(self, other)     →    self - other
__eq__(self, other)      →    self == other
__lt__(self, other)      →    self < other
__len__(self)            →    len(self)
__str__(self)            →    print self
```

... and others

matrixBi

EXAMPLE: FRACTIONS

- create a **new type** to represent a number as a fraction
- **internal representation** is two integers
  - numerator
  - denominator
- **interface** a.k.a. **methods** a.k.a **how to interact** with `Fraction` objects
  - add, subtract
  - print representation, convert to a float
  - invert the fraction
- the code for this is in the handout, check it out!

matrixBi

CLASS EXERCISE

```
a = Fraction(1,4)
b = Fraction(3,4)
c = a + b # c is a Fraction object
print(c) # 16/16
print(float(c)) # 1.0
print(Fraction.__float__(c))  # 1.0
print(float(b.inverse())) # 1.3333333

##c = Fraction(3.14, 2.7) # assertion error
##print a*b # error, did not define how to multiply two Fraction objects
```

matrixBi

THE POWER OF OOP

- **bundle together objects** that share
  - common attributes
  - procedures that operate on those attributes
- use **abstraction** to make a distinction between how to implement an object vs how to use the object
- build **layers** of object abstractions that inherit behaviors from other classes of objects
- create our **own classes of objects** on top of Python's basic classes

matrixBi

IMPLEMENTING THE CLASS vs USING THE CLASS

- write code from two different perspectives

implementing a new object type with a class
- **define** the class
- define **data attributes** (WHAT IS the object)
- define **methods** (HOW TO use the object)

using the new object type in code
- create **instances** of the object type
- do **operations** with them

matrixBi

CLASS DEFINITION OF AN OBJECT TYPE vs INSTANCE OF A CLASS

- class name is the **type**
  - `class Coordinate(object)`
- class is defined generically
  - use `self` to refer to some instance while defining the class
  - `(self.x - self.y)**2`
  - `self` is a parameter to methods in class definition
- class defines data and methods **common across all instances**

- instance is **one specific** object
  `coord = Coordinate(1,2)`
- data attribute values vary between instances
  `c1 = Coordinate(1,2)`
  `c2 = Coordinate(3,4)`
  - `c1` and `c2` have different data attribute values `c1.x` and `c2.x` because they are different objects
- instance has the **structure of the class**

matrixBi

WHY USE OOP AND CLASSES OF OBJECTS?

- mimic real life
- group different objects part of the same type


Jelly
1 year old
brown


5 years old
brown


Tiger
2 years old
brown


Bean
0 years old
black

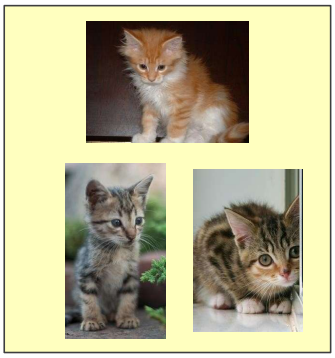

2 years old
white


1 year old
b/w

matrixBi

WHY USE OOP AND CLASSES OF OBJECTS?

- mimic real life
- group different objects part of the same type



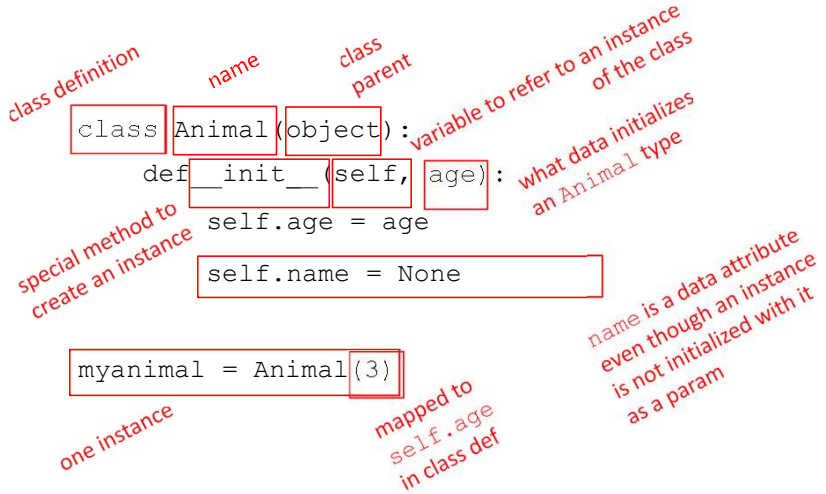
matrixBi

GROUPS OF OBJECTS HAVE ATTRIBUTES (RECAP)

- **data attributes**
  - how can you represent your object with data?
  - **what it is**
  - *for a coordinate: x and y values*
  - *for an animal: age, name*
- **procedural attributes** (behavior/operations/**methods**)
  - how can someone interact with the object?
  - **what it does**
  - *for a coordinate: find distance between two*
  - *for an animal: make a sound*

matrixBi

HOW TO DEFINE A CLASS (RECAP)



matrixBi

GETTER AND SETTER METHODS

```
class Animal(object):
    def_init_(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def_str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

getter

setter

- **getters and setters** should be used outside of class to access data attributes

matrixBi

AN INSTANCE and DOT NOTATION (RECAP)

- instantiation creates an **instance of an object**

```
a = Animal(3)
```

- **dot notation** used to access attributes (data and methods) though it is better to use getters and setters to access data attributes

```
a.age
```

- access data attribute
- allowed, but not recommended

```
a.get_age()
```

- access method
- best to use getters and setters

matrixBi

INFORMATION HIDING

- author of class definition may **change data attribute** variable names

replaced age data attribute by years

```
class Animal(object):
    def __init__(self, age):
        self.years = age
    def get_age(self):
        return self.years
```

- if you are **accessing data attributes** outside the class and class **definition changes**, may get errors
- outside of class, use getters and setters instead
  use `a.get_age()` NOT `a.age`
  - good style
  - easy to maintain code
  - prevents bugs

matrixBi

PYTHON NOT GREAT AT INFORMATION HIDING

- allows you to **access data** from outside class definition

```
print(a.age)
```

- allows you to **write to data** from outside class definition

```
a.age = 'infinite'
```

- allows you to **create data attributes** for an instance from outside class definition

```
a.size = "tiny"
```

- it's **not good style** to do any of these!

matrixBi

DEFAULT ARGUMENTS

- **default arguments** for formal parameters are used if no actual argument is given

```
def set_name(self, newname=""):
    self.name = newname
```

- default argument used here

```
a = Animal(3)
a.set_name()
print(a.get_name())
```

prints ""

- argument passed in is used here

```
a = Animal(3)
a.set_name("fluffy")
print(a.get_name())
```

prints "fluffy"

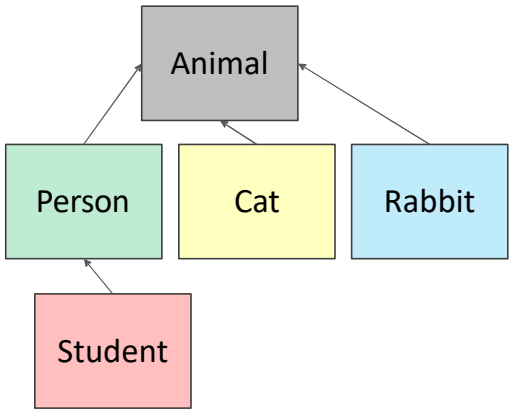
matrixBi

HIERARCHIES

matrixBi

HIERARCHIES

- **parent class** (superclass)
- **child class** (subclass)
  - **inherits** all data and behaviors of parent class
  - **add** more **info**
  - **add** more **behavior**
  - **override** behavior



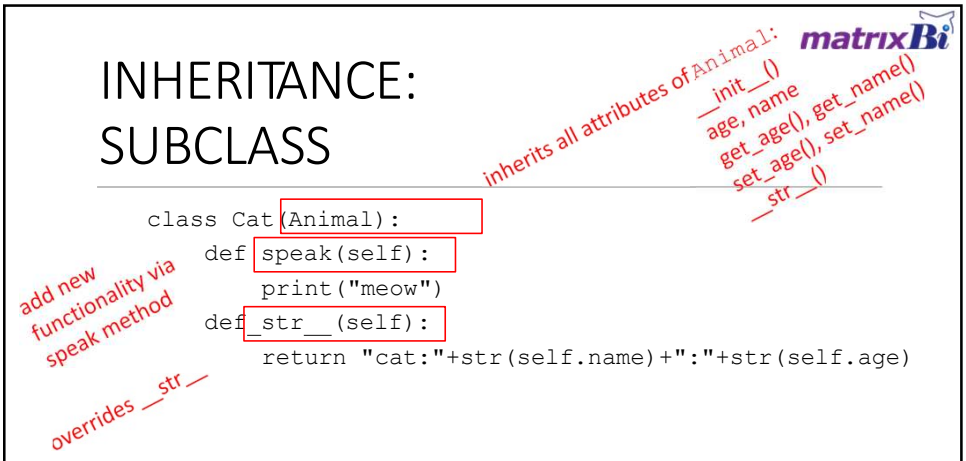
matrixBi

INHERITANCE: PARENT CLASS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

- everything is an object
- class object implements basic operations in Python, like binding variables, etc

INHERITANCE: SUBCLASS

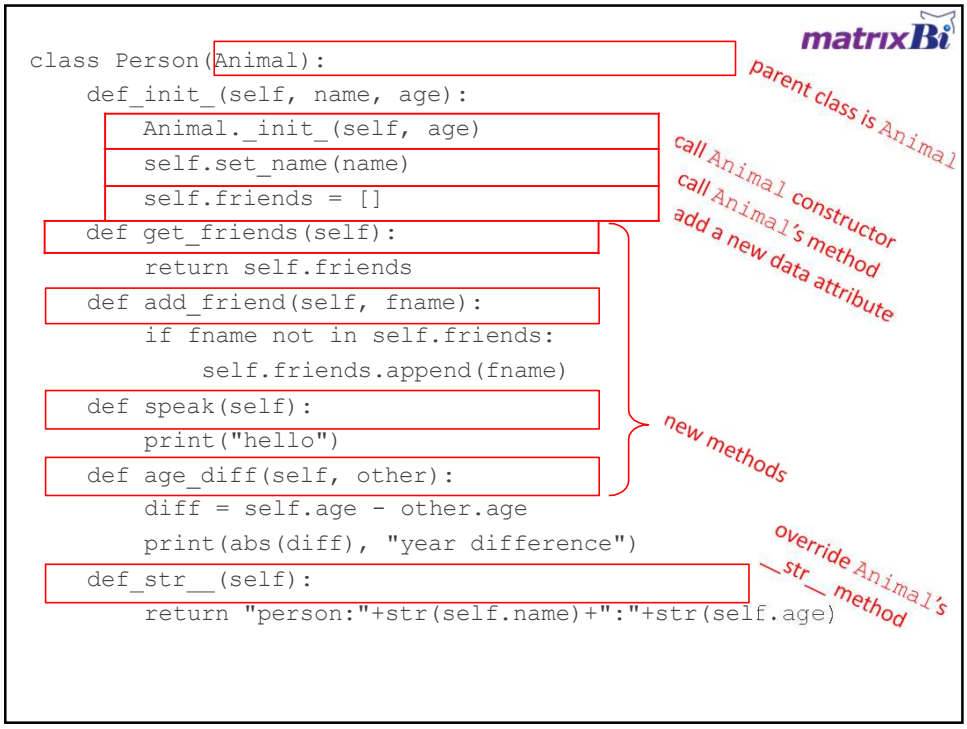


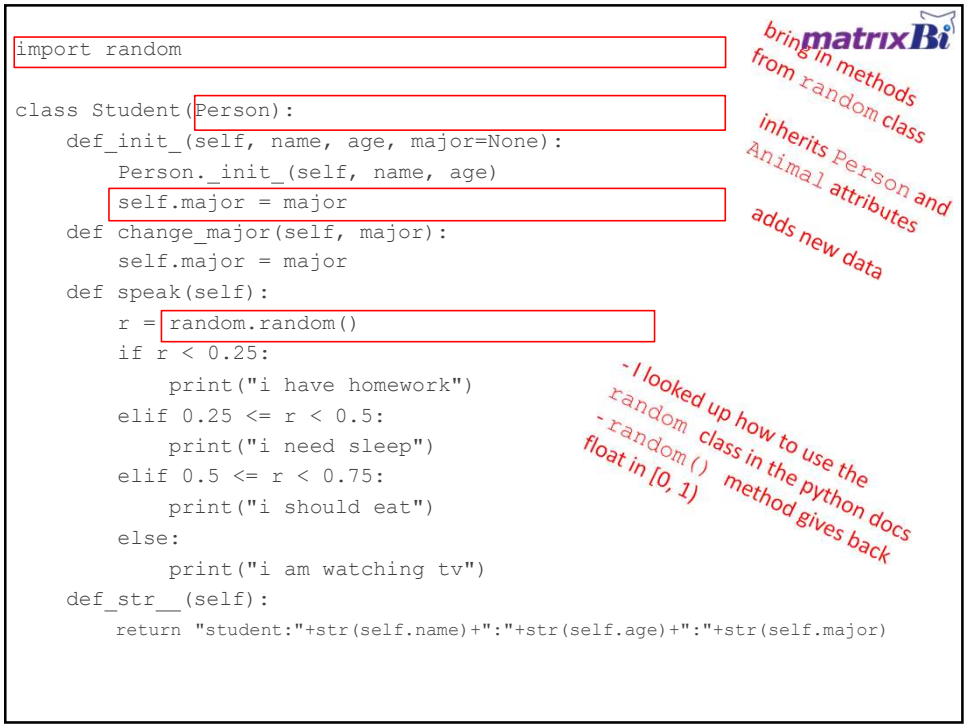
- add new functionality with `speak()`
  - instance of type `Cat` can be called with new methods
  - instance of type `Animal` throws error if called with `Cat`'s new method
- `__init__`is not missing, uses the `Animal` version

matrixBi

WHICH METHOD TO USE?

- subclass can have **methods with same name** as superclass
- for an instance of a class, look for a method name in **current class definition**
- if not found, look for method name **up the hierarchy** (in parent, then grandparent, and so on)
- use first method up the hierarchy that you found with that method name


matrixBi
class Person(Animal):
def __init__(self, name, age):
Animal.__init__(self, age)
self.set_name(name)
self.friends = []
def get_friends(self):
return self.friends
def add_friend(self, fname):
if fname not in self.friends:
self.friends.append(fname)
def speak(self):
print("hello")
def age_diff(self, other):
diff = self.age - other.age
print(abs(diff), "year difference")
def __str__(self):
return "person:"+str(self.name)+":"+str(self.age)
parent class is Animal
call Animal constructor
call Animal's method
add a new data attribute
new methods
override Animal's __str__ method


matrixBi
import random
class Student(Person):
def __init__(self, name, age, major=None):
Person.__init__(self, name, age)
self.major = major
def change_major(self, major):
self.major = major
def speak(self):
r = random.random()
if r < 0.25:
print("i have homework")
elif 0.25 <= r < 0.5:
print("i need sleep")
elif 0.5 <= r < 0.75:
print("i should eat")
else:
print("i am watching tv")
def __str__(self):
return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
bring in methods from random class
inherits Person and Animal attributes
adds new data
- I looked up how to use the random class in the python docs
- random() method gives back float in [0, 1)

matrixBi

CLASS VARIABLES AND THE `Rabbit` SUBCLASS

- **class variables** and their values are shared between all instances of a class

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

parent class

class variable

access class variable

instance variable

incrementing class variable changes it for all instances that may reference it

- `tag` used to give **unique id** to each new rabbit instance

matrixBi

`Rabbit` GETTER METHODS

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
    def get_rid(self):
        return str(self.rid).zfill(3)
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
```

method on a string to pad the beginning with zeros for example, 001 not 1

- getter methods specific for a `Rabbit` class
- there are also getters `get_name` and `get_age` inherited from `Animal`

matrixBi

WORKING WITH YOUR OWN TYPES

```
def __add__(self, other):
        # returning object of same type as this class
        return Rabbit(0, self, other)
```

recall Rabbit's `__init__(self, age, parent1=None, parent2=None)`

- define **+ operator** between two `Rabbit` instances
  - define what something like this does: `r4 = r1 + r2` where `r1` and `r2` are `Rabbit` instances
  - `r4` is a new `Rabbit` instance with age 0
  - `r4` has `self` as one parent and `other` as the other parent
  - in `__init__`, **`parent1` and `parent2` are of type `Rabbit`**

matrixBi

SPECIAL METHOD TO COMPARE TWO `Rabbits`

- decide that two rabbits are equal if they have the **same two parents**

```
def __eq__(self, other):
    parents_same = self.parent1.rid == other.parent1.rid \  and
self.parent2.rid == other.parent2.rid
    parents_opposite = self.parent2.rid == other.parent1.rid \ and
    self.parent1.rid == other.parent2.rid  return parents_same or
    parents_opposite
```

booleans

- compare ids of parents since **ids are unique** (due to class var)
- note you can't compare objects directly
  - for ex. with `self.parent1 == other.parent1`
  - this calls the `__eq__` method over and over until call it on `None` and gives an `AttributeError` when it tries to do `None.parent1`

matrixBi

OBJECT ORIENTED PROGRAMMING

- create your own **collections of data**
- **organize** information
- **division** of work
- access information in a **consistent** manner
- add **layers** of complexity
- like functions, classes are a mechanism for **decomposition** and **abstraction** in programming

matrixBi

Python naming convention

Type	Public	Internal
Packages	`lower_with_under`	
Modules	`lower_with_under`	`_lower_with_under`
Classes	`CapWords`	`_CapWords`
Exceptions	`CapWords`	
Functions	`lower_with_under()`	`_lower_with_under()`
Global/Class Constants	`CAPS_WITH_UNDER`	`_CAPS_WITH_UNDER`
Global/Class Variables	`lower_with_under`	`_lower_with_under`
Instance Variables	`lower_with_under`	`_lower_with_under`
Method Names	`lower_with_under()`	`_lower_with_under()`
Function/Method Parameters	`lower_with_under`	
Local Variables	`lower_with_under`	

matrixBi

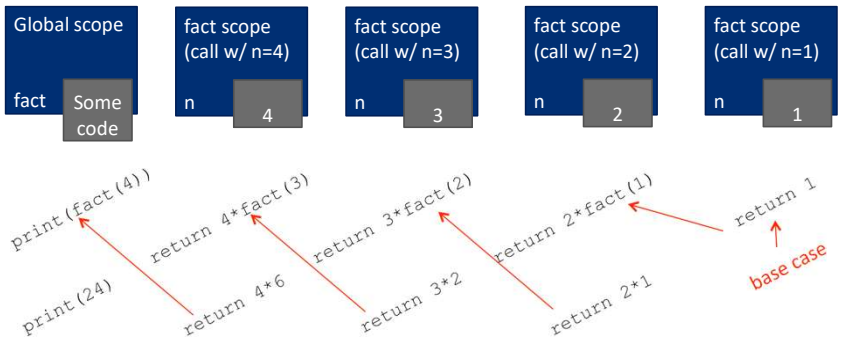
WHAT IS RECURSION?

- Algorithmically: a way to design solutions to problems by **divide-and-conquer** or **decrease-and-conquer**
  - reduce a problem to simpler versions of the same problem
- Semantically: a programming technique where a **function calls itself**
  - in programming, goal is to NOT have infinite recursion
    - must have **1 or more base cases** that are easy to solve
    - must solve the same problem on **some other input** with the goal of simplifying the larger problem input

matrixBi

RECURSIVE FUNCTION SCOPE EXAMPLE

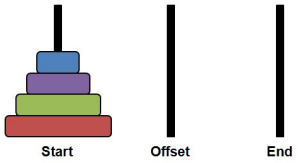
```
def fact(n):
    if n == 1:
        return 1
    else:
        return n*fact(n-1)

print(fact(4))
```



matrixBi

TOWERS OF HANOI

- The story:
  - 3 tall spikes
  - Stack of 64 different sized discs – start on one spike
  - Need to move stack to second spike (at which point universe ends)
  - Can only move one disc at a time, and a larger disc can never cover up a small disc

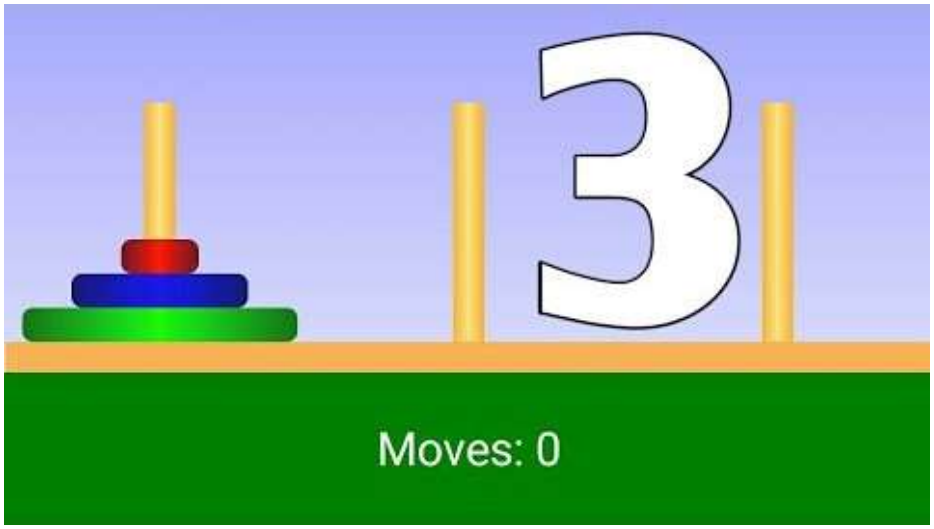


matrixBi

TOWERS OF HANOI

- Having seen a set of examples of different sized stacks, how would you write a program to print out the right set of moves?
- **Think recursively!**
  - Solve a smaller problem
  - Solve a basic problem
  - Solve a smaller problem

matrixBi



matrixBi

```
def printMove(fr, to):
    print('move from ' + str(fr) + ' to ' +
    str(to))

def Towers(n, fr, to, spare):
    if n == 1:
        printMove(fr, to)
    else:
        Towers(n-1, fr, spare, to)
        Towers(1, fr, to, spare)
        Towers(n-1, spare, to, fr)
```

PRACTICE

- תרגילים בסיסיים בתכנות