Crash course in Python

Basic data types

Numbers: integers and floating point

int

```
number_of_students = 47
print(type(number_of_students))
```

47 is data of type int

The value associated with the variable number_of_students is of type int

float

```
purchase_price = 93.74
print(type(purchase_price))
```

93.74 is data of type float

Operations with numbers

multiplication: `x * y`

division: `x/y`
  always returns float

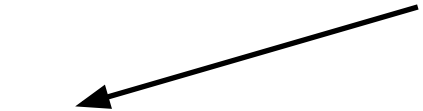
integer division: `x//y`
  returns truncated int (as int or float)

remainder: `x%y`
  returns remainder (as int or float)

power: `x ** y`

strings: str

John is a string. Python does not differentiate the meaning of “ and ‘



```
x="John"
x='John'
print(type(x))
```

string operations

Always take a banana to a party is a string literal, i.e., an actual value

```
x="Always take a banana to a party!"
```

* A string is an ordered collection of characters
* Location matters. We can access characters by location

```
y=x[0]  #The value of y is 'A'
y=x[3]  #The value of y is 'a'
y=x[-1] #The value of y is '!'
y=x[32] #IndexError! (out of range)
len(x) #Returns the number of characters in x
```

string operations

x="Always take a banana to a party!"

We can extract substrings from a string

```
y=x[7:11] #The value of y is 'take' (locations 7, 8, 9, 10)
y=x[7:] #The value of y is 'take a banana to a party!'
y=x[0::2] #The value of y is 'Awy aeabnn oapry' (every 2nd character
y=x[::-1] #The value of y is ???? (what does the negative sign mean?)
```

string operations

x="Always take a banana to a party!"

the find function returns the location of a substring in a string

y=x.find("to") # The value of y is 21 (find returns the first instance)
y=x.find("hello") # -1 (indicates that the substring was not found

strings are immutable

```
x="Always take a banana to a party!"
```

the value of a string cannot be changed

```
x[5]='C' #TypeError! (string objects are not changeable)
```

String concatenation

```
x="Always take a banana to a party!"
y=" Never forget"
z = x+y
print(z)
```

the value of y is added at the end of the value of x and the entire result is stored in the new string z

Variables and values

Values do not change but the value associated with a variable can change!

```
In [8]: student1 = 87
        id(student1)
Out[8]: 4297151280

In [9]: student2 = 95
        id(student2)
Out[9]: 4297151536

In [10]: student1 = student2
         id(student1)
Out[10]: 4297151536

In [11]: id(87)
Out[11]: 4297151280
```

id: a function that returns the location of a value in memory

student1: its value is at the same location where student2's value is

87: its location in memory is unchanged

bool

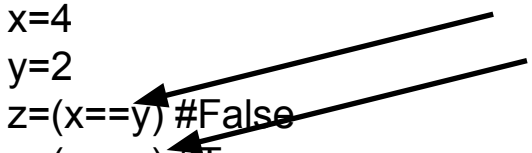
Syntax note: uppercase T followed by lowercase rue - nothing else is True!
(likewise for False)

bool

```
x=4
y=2
z=(x==y) #False
z=(x==x) #True
a=True
b=False
```

z takes the value False

the value of z changes to True



bool: Relational and Logical operators

<	x < y	True if x is less than y
>	x > y	True if x is greater than y
<=	x<=y	True if x is less than or equal to y
>=	x>=y	True if x is greater than or equal to y
not	not x	True if x is False
and	x and y	True if both x and y are True
or	x or y	True if either x is True or y is True or both are True

bool

In python, everything has a truth value

Anything that evaluates to 0 or nothing is False
Anything that is non-zero or something is True

x=8
print(bool(x)) —-> True

y=‘’
print(bool(y)) ——> False

print(x==y) —-> False #already bool so no conversion necessary

bool

The truth value and actual value of an expression are not the same thing

```
x=8
print(bool(x)) —-> True #But x is still 8

y=‘’
print(bool(y)) ——> False #But y is still an empty string

z = 43.4
print(bool(z))       —-> True #But z is still 43.4

p=(x==z)     —-> False #Because x==z is a relational operator
#Relational operators always evaluate to True or False

result = x and z —->
            #First x is evaluated and its boolean value is True
            #Then z is evaluated and its boolean value is True
            #Since z is the last value evaluated, the expression
                returns 43.4
```

bool

Logical expressions are evaluated only to the extent necessary to determine their truth value (**lazy evaluation**)

x=8
y=0
result = x and y #0 because y is evaluated last
result = x or y #8 because if x is True then y doesn't matter
result = y and x #0 because y is False and x doesn't matter

variables and assignment

variables must be declared before you can use them!

Almost always by placing the variable name on the left hand side of an **assignment statement**

Examples

```
price_now = float(input("What is the price now?"))
pct_return = (price_now - initial_price)/initial_price *100
print("The return on the stock is: ",pct_return)
```

variables and assignment

assignment statements assign values to variables

the left hand side of an assignment statement is (almost!) ALWAYS a single variable name

the right hand side of an assignment statement MUST resolve to a value

variables and assignment

```
x = 5  #Simple assignment

x = y = 5 #Multiple assignment

x,y = 3,4 #Unpacking assignment

x += 4  #Augmented assignment
```

the if statement

The “if” statement and logical expressions

Logical expressions are used to control program flow

Consider a simple trading strategy:

1. If the price of a stock drops more than 10% below the cost basis - close the position as a STOP LOSS
2. If the price of the stock goes up by more than 20% - close the position as PROFIT TAKING
3. If neither 1 nor 2 work, then do nothing

program control flow

```
purchase_price = float(input("Purchase price? "))
price_now = float(input("Price now? "))
if price_now < .9 * purchase_price:
    print("Stop Loss activated. Close the position”)
elif price_now > 1.2 * purchase_price:
    print("Profit taking activated. Close the position")
else:
    print("Do nothing")
```

this gets done only if neither logical expression evaluates to True

this gets done only if the first logical expression is False and the second one is True

if … elif …. else ….

```
if condition1 :
   statement1_1
   statement1_2
   …
   statement1_n
elif condition2 :
   statement2_1
   statement2_2
   …
   statement2_n
elif condition3 :
   statement3_1
   statement3_2
   …
   statement3_n
else:
   statement4_1
   statement4_2
   …
   statement4_n
post_if_statement1
post_if_statement2
……
```

if condition 1 is True then the program does statements 1_1 to 1_n and jumps to the post_if statements

if condition 1 is False then condition 2 is checked. If it is True, then the program does statements 2_1 to 2_n and jumps to the post_if statements

And so on

If neither of the if or elif conditions evaluate to True then, and only then, statements 4_1 to 4_n are executed

Syntax note: program blocks

```
purchase_price = float(input("Enter the purchase price of the stock: "))
price_now = float(input("Enter the current price of the stock: "))
if price_now < purchase_price * 0.9:
    print("STOP LOSS: Sell the stock! ")
    print("You've lost",purchase_price-price_now,"Dollars per share")
elif price_now > purchase_price * 1.2:
    print("PROFIT TAKING: Sell the stock!")
    print("You've gained",price_now-purchase_price,"Dollars per share")

else:
    print("HOLD: Don't do anything!")
    print("Your unrealized profit is",price_now-purchase_price,"Dollars per share")
print("Hope you enjoyed this program!")
```

this is a block. note the indenting!

A colon indicates that a block will follow

the end of indenting indicates that the block has ended

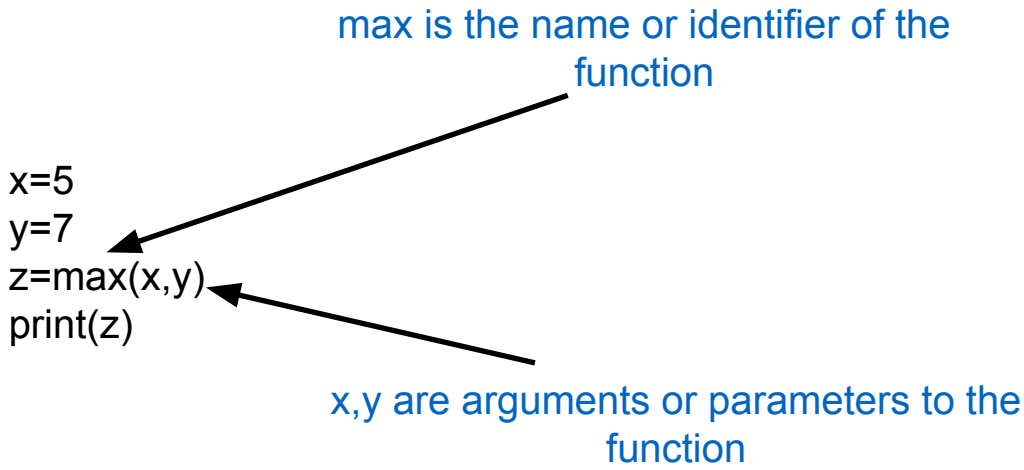
Nested blocks

```
purchase_price = float(input("Purchase price? "))
price_now = float(input("Price now? "))
days_held = int(input("Number of days position held? "))
if price_now < .9 * purchase_price:
    if days_held < 10:
        if price_now < .8 * purchase_price:
            print("Stop Loss Activated. Close the position")
        else:
            print("Do nothing")
    else:
        print("Stop Loss activated. Close the position")
elif price_now > 1.1 * purchase_price:
    print("Profit taking activated. Close the position")
else:
    print("Do nothing")
```

this is a nested block. note the additional indenting!

Functions

Calling a function

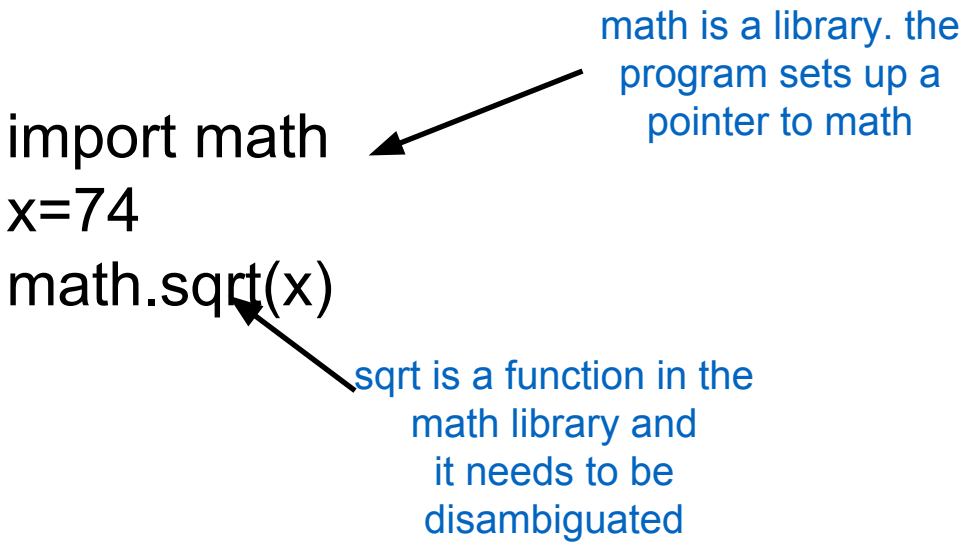


max is a black box. we don't know how python is figuring out which one is the greater of the two (and we don't want to know!)

Function libraries

Functions can be grouped in libraries

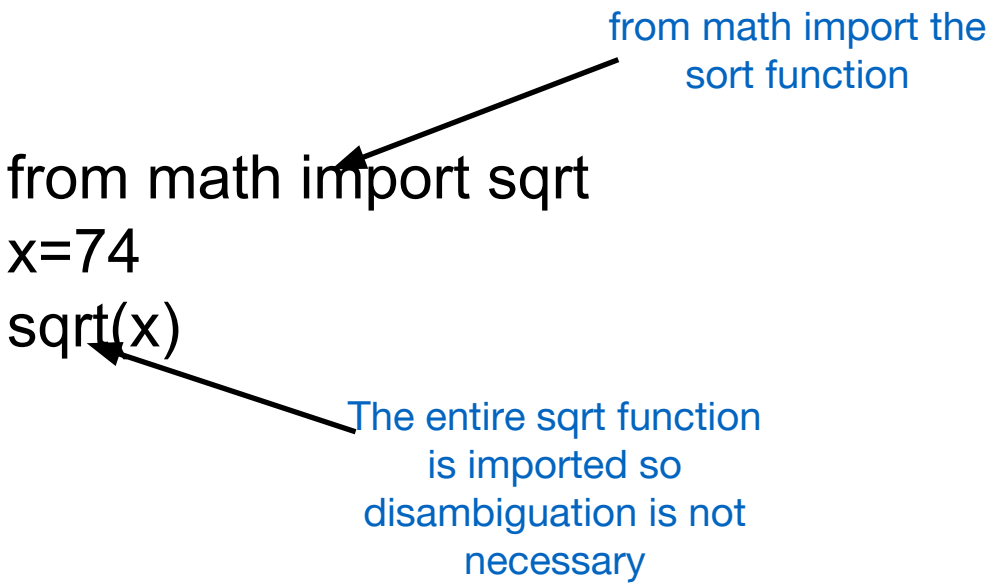
Libraries need to be imported into a program



Function libraries

Functions can be grouped in libraries

Libraries need to be imported into a program



Function libraries

Python is an open source language

With many libraries

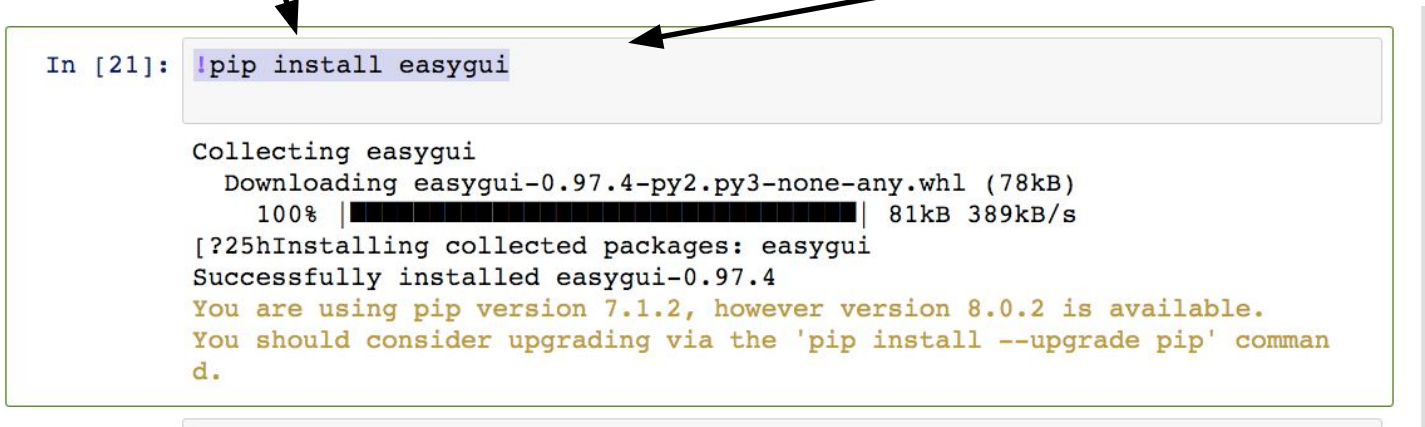
Most need to be explicitly installed on your computer

Authenticated libraries are available at
https://pypi.python.org/pypi

Install libraries using pip

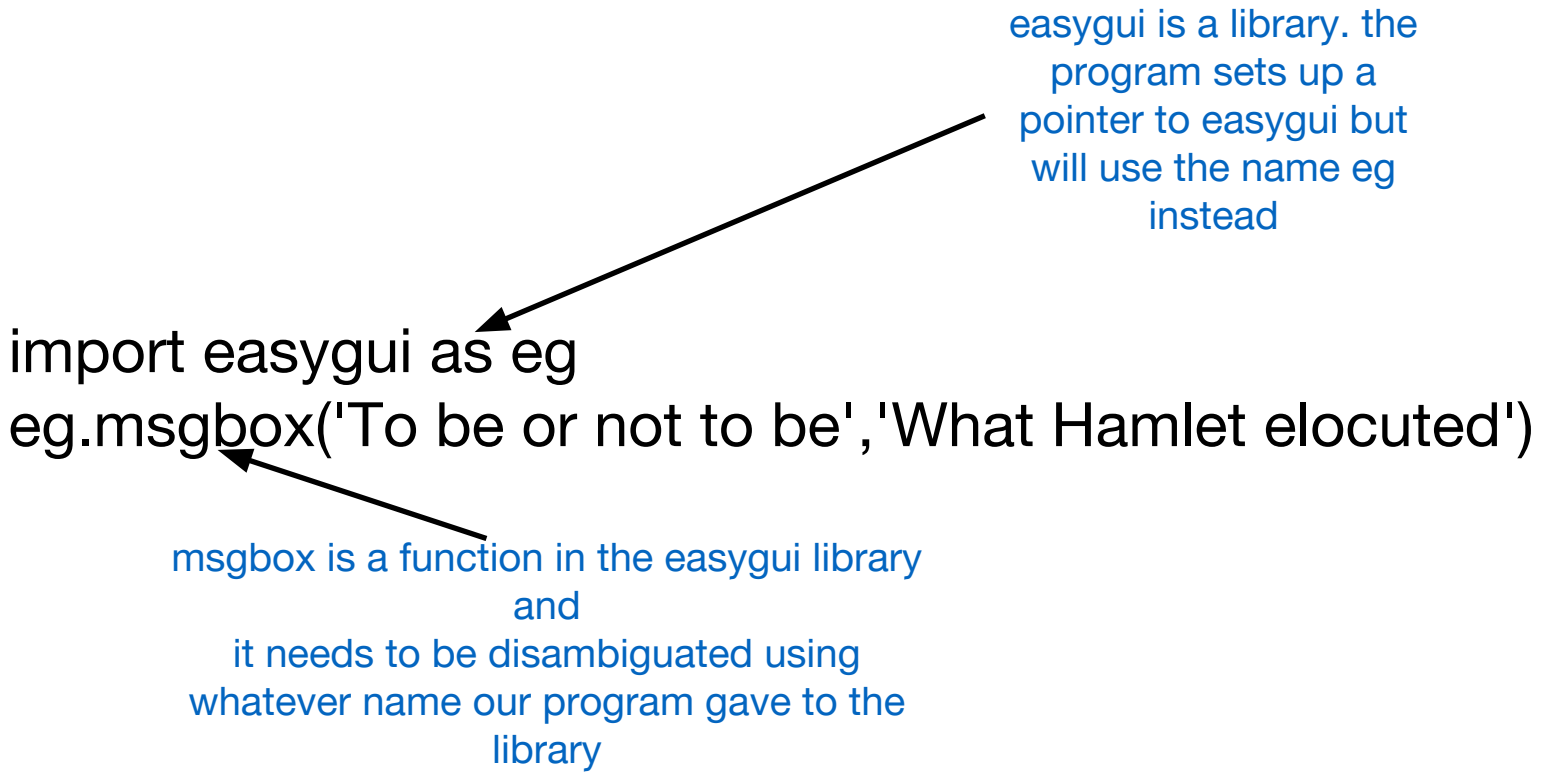
pip: python installer program

easygui: a gui development library

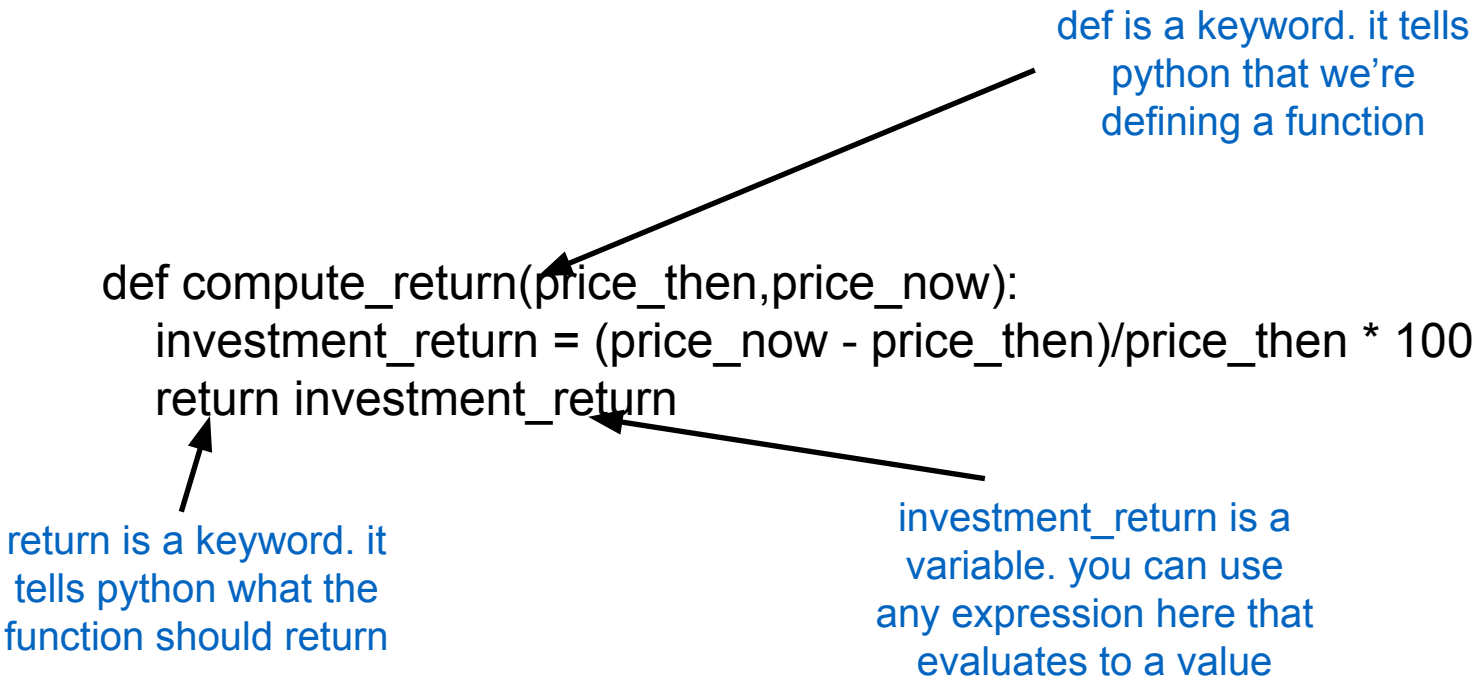


pip is an independent program and can be run directly from windows powershell or mac's terminal. Anaconda ipython notebook is the hassle free way of installing libraries

And them import them into your program



Defining your own functions



Returning values from a function

A function returns a value through the return statement. If there is no return statement, python uses None

```
def spam(x):
  x=x+1

print(spam(5)) —-> None
```

Returning multiple values from a function

```
def minmax(x,y):
    return min(x,y),max(x,y)
x,y = minmax(7,2)
print(x,y) —-> 2,7
```

multiple assignment. x will take the value of the first item on the RHS and y the second. The RHS items must be separated by commas

Passing arguments to a function

arguments are assigned values from left to right

```
def div(x,y):
   return x/y

a=30
print(div(a,10)) ——-> x is 30, y is 10, prints 3
```

```
def div(x,y):
   return x/y

x=10
y=30
print(div(y,x)) ——-> x is 30, y is 10, prints 3
```

Passing arguments to a function

You can give values to arguments directly in a function call

```
def div(x,y):
   return x/y

print(div(x=30,y=10)) —-> 3
print(div(y=10,x=30)) —-> 3
```

Functions can have default arguments

```
def compute_return(x,y,z=0):
    investment_return=(y-x)/x
    if z and z==100:
        investment_return * 100
    return investment_return
```

0 is the default for z

z is 0

```
r1 = compute_return(1.2,91.2)

r1 = compute_return(1.2,91.2,100)
```

z is 100

Functions can have functions as arguments (first order functions)

```
def order_by(a,b,order):
    return order(a,b)

order_by(4,7,max)
```

since we're using order like a function, it must be a function

pass the function max to order_by

Collections and Iteration

Collections

Lists: Sequential ordered mutable collections

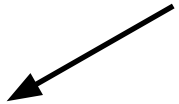
Key properties

* collection of related objects
* ordered or sequential collection
* mutable. Lists can be modified

Examples

```
list_of_names = ["John","Jack","Jill","Joan"]
list_of_tickers = [“AAPL","IONS","GE","DB"]
list_of_natural_numbers = [1,2,3,4,5,6,7]
long_list = [1,['a',['b','c']],43,"Too many cooks spoil the broth"]
```

objects in a list don’t have to be of the same type



Operations on lists

```
long_list = [1,['a',['b','c']],43,"Too many cooks spoil the broth"]
long_list.append('Many hands make light work') #adds an item to the back of the list
long_list[3] #Gets the 4th item in the list
long_list[1][1][0] #Accessing nested items
long_list.extend(['e','f']) #appends contents of a list
long_list.remove(1) #Removes the item with the VALUE 1
long_list.pop() #Removes and returns the last item
long_list.pop(1) #Removes and returns the ith item
len(long_list) #Returns the length of the list
```

Lists are mutable

Contents of a list can be changed

Examples

x = [1,2,3,4]
x[0]=8 —-> [8,2,3,4]

Mutable vs immutable

immutable: data objects that cannot be changed
e.g. the number 5 is immutable (we can't make it into an 8!)

mutable: data objects that can be changed
e.g., a list of objects owned by Jack and Jill
['pail','water']
(it can be changed to ['pail'])

int, str, bool, float are immutable
list objects are mutable
every python object is either mutable or immutable

Mutable vs immutable

Try this?

```
x = [1,2,3]
y = x
x[2] = 4
print(x)
print(y)
```

Mutable vs immutable

And this

```
y=['a','b']
x = [1,y,3]
y[1] = 4
print(x)
print(y)
```

Mutable vs immutable
What's the difference?

```
def eggs(item,total=0):
    total+=item
    return total
print(eggs(1))
print(eggs(2))
```

```
def spam(elem,some_list=[]):
    some_list.append(elem)
    return some_list
print(spam(1))
print(spam(2))
```

Tuples: sequential, immutable collections

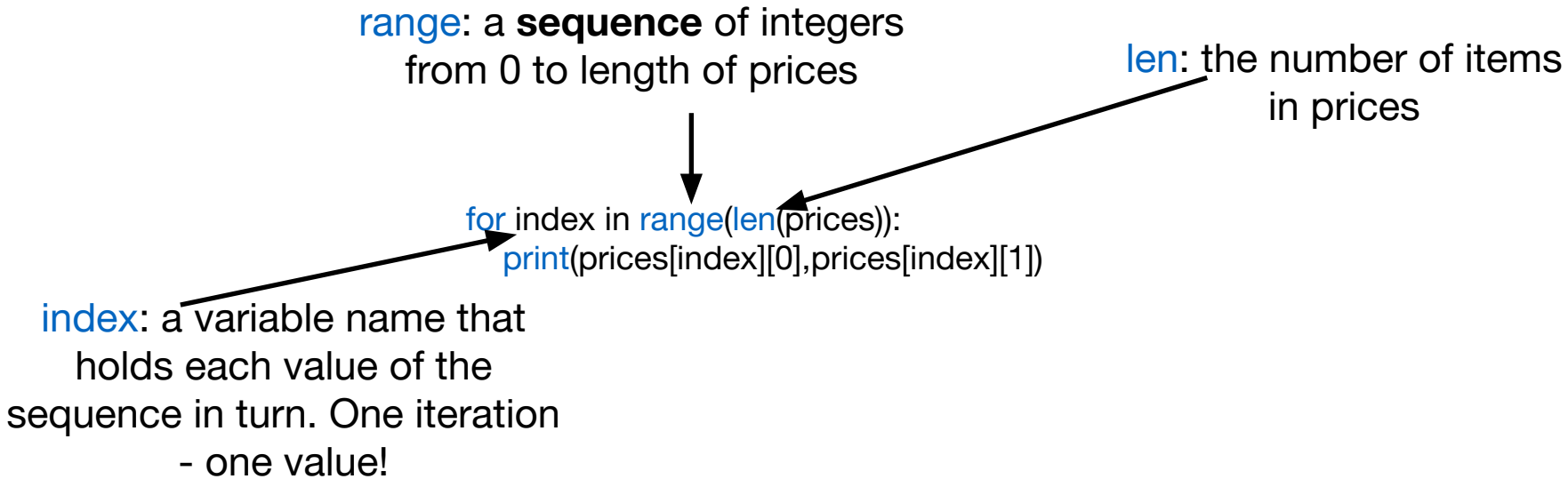
price = ("20150904",545.23)
price[0] —> "20140904"
price[1] —-> 545.23
price[1]=26.3 —-> TypeError
price[2] —-> IndexError

Tuples are just like lists except they are not mutable (cannot be changed)

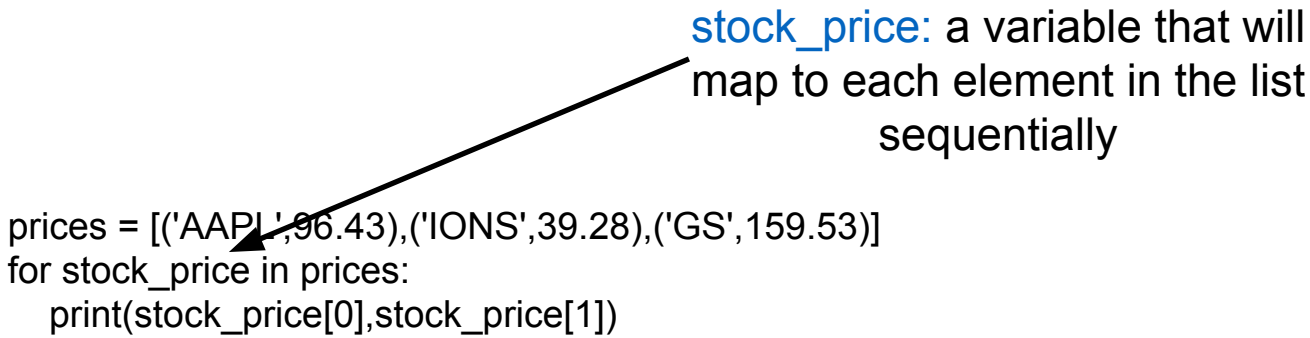
All list operations, except for the ones that change the value of a list, are also valid tuple operations

Iteration

iterating using location indices



iterating by accessing items sequentially



controlling iteration: break and else

```
prices = [('AAPL',96.43),('IONS',39.28),('GS',159.53)]
ticker = input('Please enter a ticker: ')
for item in prices:
    if item[0] == ticker:
        print(ticker,item[1])
        break
else:
    print("Sorry",ticker,"was not found in my database")
print("Statement after for")
```

the for block

break: the loop will end and control will pass outside the for loop

else: the program will do this only if the for does not encounter a 'break'

practice problem

Write a function search_list that searches a list of tuple pairs and returns the value associated with the first element of the pair

```
prices = [('AAPL',96.43),('IONS',39.28),('GS',159.53)]
x=search_list(prices,'AAPL')
#The value of x should be 96.43
x=search_list(prices,'GOOG')
#The value of x should be None

inventory = [('widgets',100),('spam',30),('eggs',200)]
y=search_list(inventory,'spam')
#The value of y should be 30
y=search_list(prices,'hay')
#The value of y should be None
```

Dictionaries: key-value pairs

```
mktcaps = {'AAPL':538.7,'GOOG':68.7,'IONS':4.6}
mktcaps['AAPL']       #key-based retrieval
print(mktcaps['AAPL'])
mktcaps['GE'] #error (no "GE")
'GE' in      mktcaps
mktcaps.keys() #returns a list of keys
sorted(mktcaps.keys()) #returns a sorted list of keys
```

Sets: unordered collections of unique objects

```
tickers={"AAPL","GE","NFLX","IONS"}
regions={"North East","South","West coast","Mid-West"}
"AAPL" in tickers #membership test
"IBM" not in tickers #non-membership test
pharma_tickers={"IONS","IMCL"}
tickers.isdisjoint(pharma_tickers) #empty intersection
pharma_tickers <= tickers #subset test
pharma_tickers <      tickers      #proper-subset test
tickers      > pharma_tickers #superset
tickers      & pharma_tickers #intersection
tickers      | pharma_tickers #union
tickers      - pharma_tickers #set difference
```