Basic Python

Amitabha Sanyal

What is Python used for?

- Web-development
  - Example packages: request, django, flask, twisted, beautifulsoup, selenium.
- Data Science
  - Example packages: numpy, pandas, matplotlib, nltk, opencv
- ML & AI
  - Example packages: Tensorflow, Pytorch, Keras, Scikit-learn

`This video` gives a brief introduction.

Python Installation

- Any version higher than 3.5 is ok. I use 3.7

```
> python --version
> sudo apt-get update
> sudo apt-get install python3.7
```

- Python's package manager called **pip** should be automatically installed. Upgrade it using:

```
> sudo pip install -U pip
```

- Now you are ready to go.

Python features

We shall point out the differences with C++.

- Python is an interpreted language and not a compiled language.

  Consider the Collatz conjecture. Define a function f defined as

```
f (n) = n/2,    if n is even
      = 3n+1,   if n is odd
```

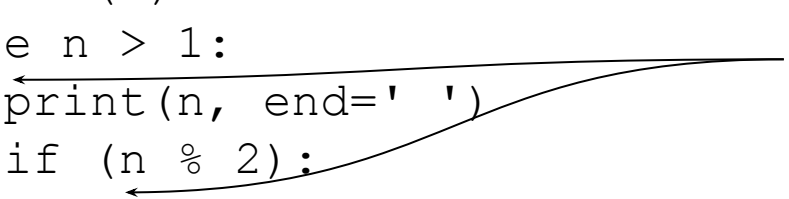
  The conjecture is that f(n) always terminates with the value 1.

- We want to test this conjecture.

Python Example

We just developed the program:

```
def collatz(n):
    while n > 1:
        print(n, end=' ')
        if (n % 2):
            # n is odd
            n = 3*n + 1
        else:
            # n is even
            n = n//2
    print(1)
n = int(input('Enter n: '))
print('Sequence: ', end='')
collatz(n)
```



Block definition by indentation

Python Example

We just developed the program:

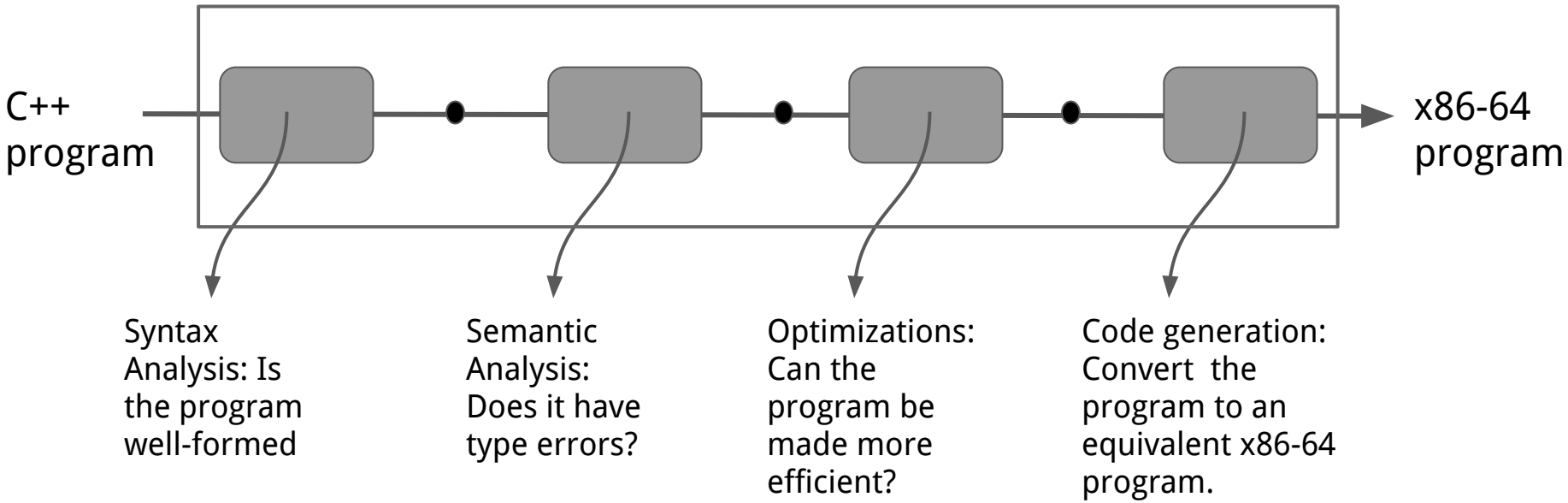
```
def collatz(n):
    while n > 1:
        print(n, end=' ')
        if (n % 2):
            # n is odd
            n = 3*n + 1
        else:
            # n is even
            n = n//2
    print(1)
n = int(input('Enter n: '))
print('Sequence: ', end='')
collatz(n)
```

- When executed, produces the result.
- Does not produce an executable as in C++.

  $ gcc -o collatz collatz.cpp

Interpretation vs Compilation

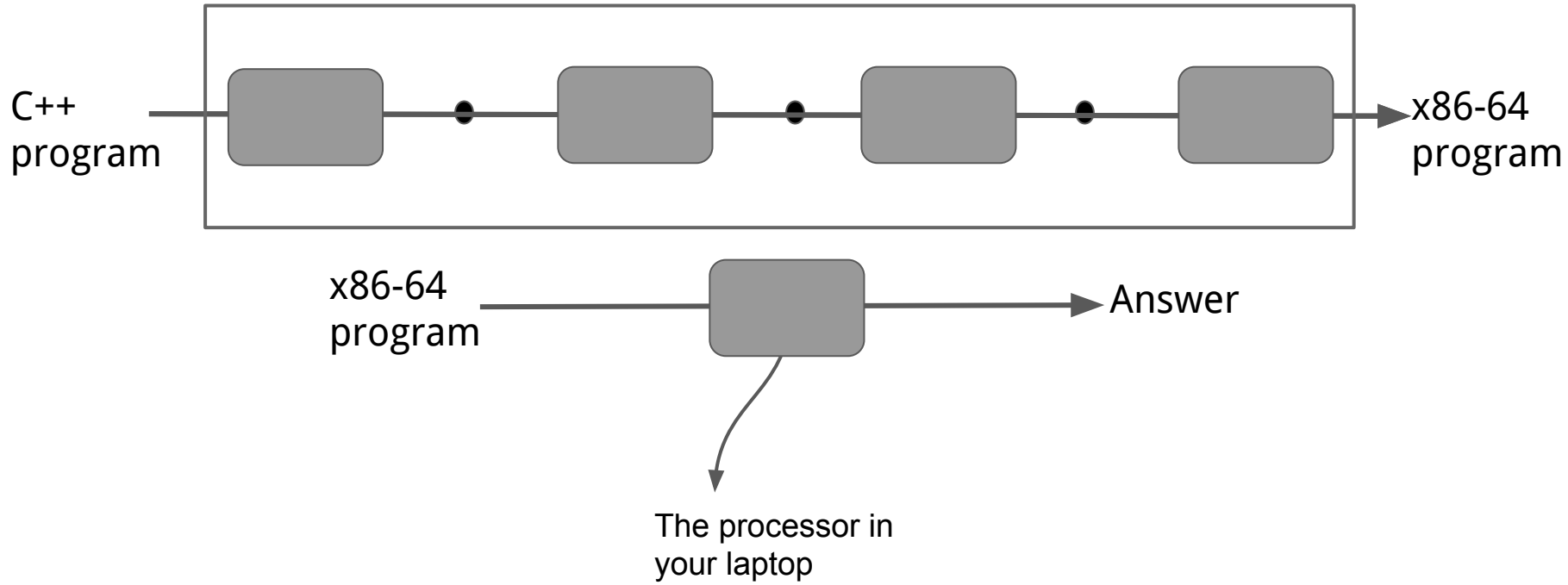
- The story of a C++ compiler for x86-64 (gcc)



- Warning: Oversimplified model. Many details omitted!

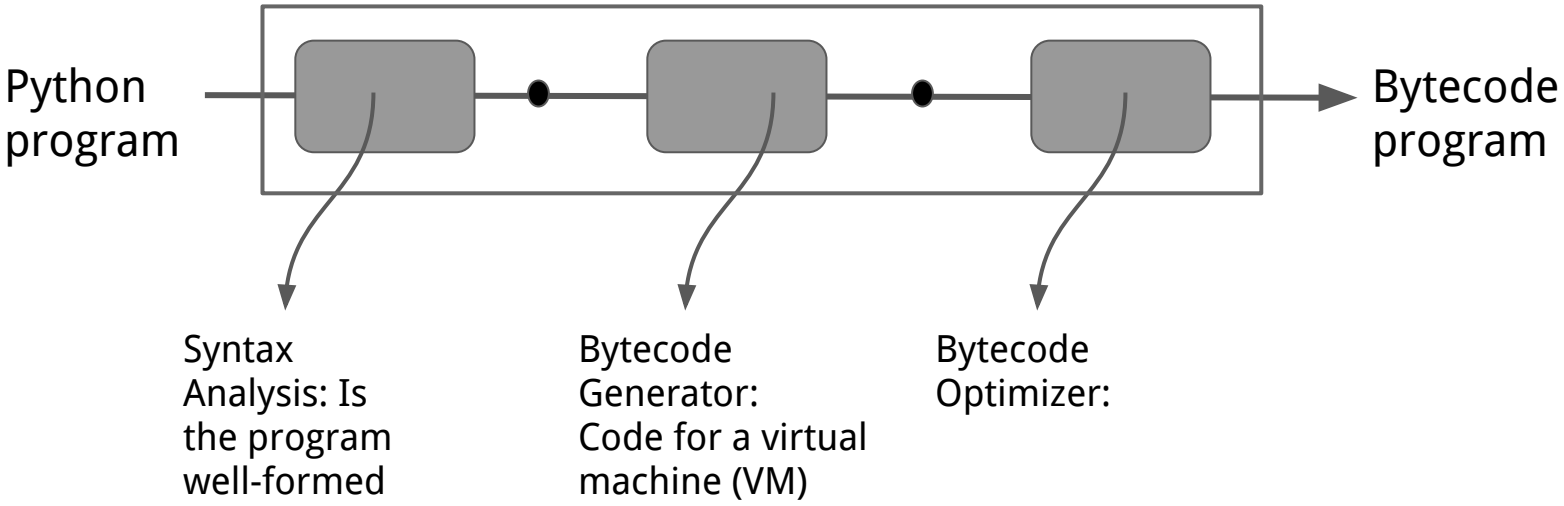
Interpretation vs Compilation

- The story of a C++ compiler for x86-64



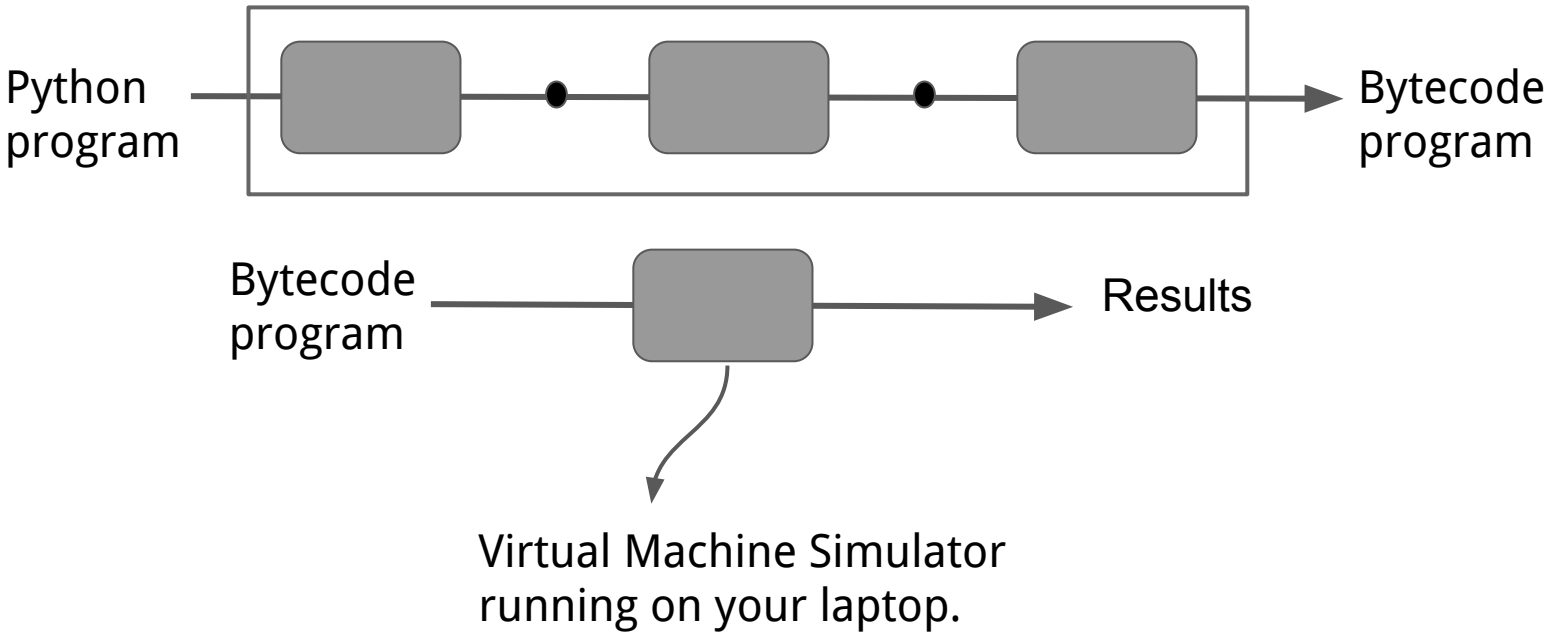
Interpretation vs Compilation

- The story of a Python interpreter for x86-64



Python Resources

- The story of a Python interpreter (CPython)

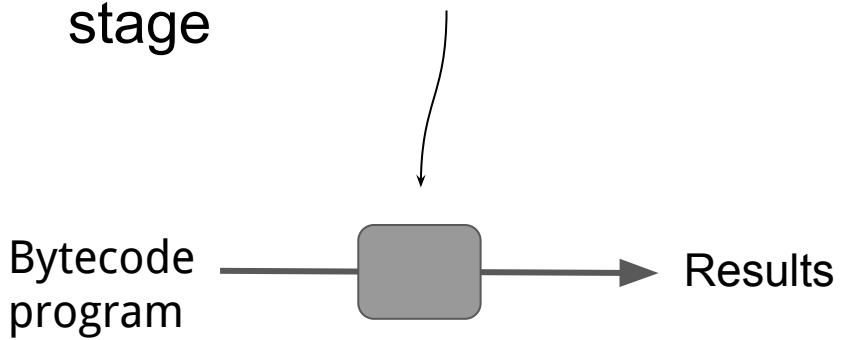


Python has dynamic type checking

```
def fact(n):
    if (n < 0):
        return "The argument
cannot be negative"
    elif (n == 0):
        return 1
    else:
        return (n*fact (n-1))

arg = int(input("type in a
number: "))
print(4+fact(arg))
```

- This program does not give a compile time error.
- Gives a runtime error only if n is less than 0.
- Type checking done at this stage



Virtual Machine Simulator running on your laptop.

Bytecode and type checking

```
1   import dis
    ...
4   def collatz(n):
5     while n > 1:
6         print(n, end=' ')
7         if (n % 2):
8             # n is odd
9             n = 3*n + 1
10        else:
11            # n is even
12            n = n//2
13    print(1)
14  def main ():
15    dis.dis(collatz)
```

```
 5           0 SETUP_LOOP              54 (to 56)
       >>    2 LOAD_FAST                0 (n)
             4 LOAD_CONST               1 (1)
             6 COMPARE_OP               4 (>)
             8 POP_JUMP_IF_FALSE       54

 6          10 LOAD_GLOBAL              0 (print)
            12 LOAD_FAST                0 (n)
            14 LOAD_CONST               2 (' ')
            16 LOAD_CONST               3 (('end',))
            18 CALL_FUNCTION_KW         2
            20 POP_TOP

 9          30 LOAD_CONST               5 (3
            32 LOAD_FAST                0 (n
            34 BINARY_MULTIPLY
            36 LOAD_CONST               1 (1
            38 BINARY_ADD
            40 STORE_FAST               0 (n
            42 JUMP_ABSOLUTE
```

Bytecode and type checking

- Type checking is part of the BINARY_ADD and BINARY_MULTIPLY instructions
- For a deep dive into the entire compilation procedure, have a look inside the virtual machine.
- The details of an operator such as BINARY_ADD is available at this site.

Running a python program

- We can use the file example.py **as a "script"**--a standalone program.
  - From the command line ...

```
$  python examples.py
```

  - ...or from the shell

```
»  exec(open('test.py').read())
```

- Or we can import the program **as a "module"** into the python shell

```
»  import example
```

  - for reloading the same module, import does not work, instead:

```
»  import importlib
»  importlib.reload(example)
```

- The variable __name__ distinguishes the use of examples.py as script or module.

```
def collatz(n):
    while n > 1:
        print(n, end=' ')
        if (n % 2):
            n = 3*n + 1
        else:
            n = n//2
    print(1)


def main ():
    n = int(input('Enter n: '))
    print('Sequence: ', end='')
    collatz(n)


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

Running a python program

- Alternately, we can import the module example into another file, say `test.py`:

```
import example as ex

n = int(input('Enter n: '))
print('Sequence: ', end='')
ex.collatz(n)
```

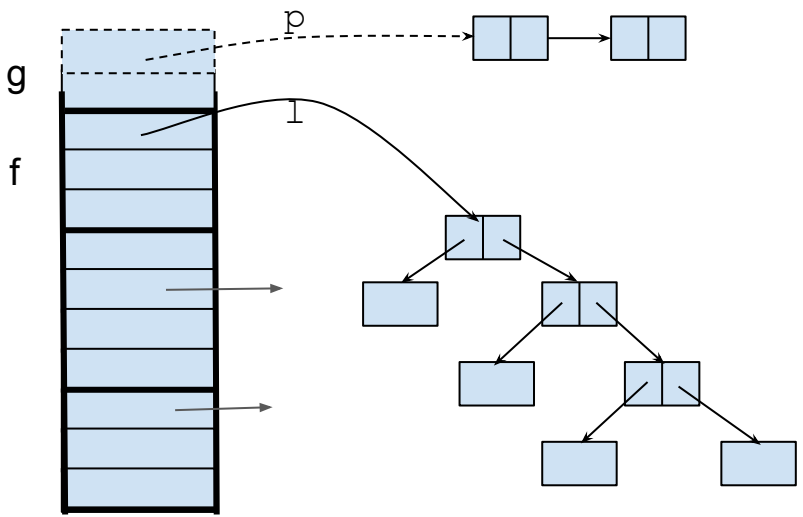
Python has Automated Garbage Collection

- Memory is allocated in two places? Stack and Heap
- Memory allocation in C++

```
void f (int i, int j)
 {
   lptr l;
   create a list and assign to l;
   ...
   g(i)
   ...
   runs out of memory
}

void g (int k)
{
  lptr p;
  create a list and assign to p;
  use p;

  programmer forgets to free p
}
```



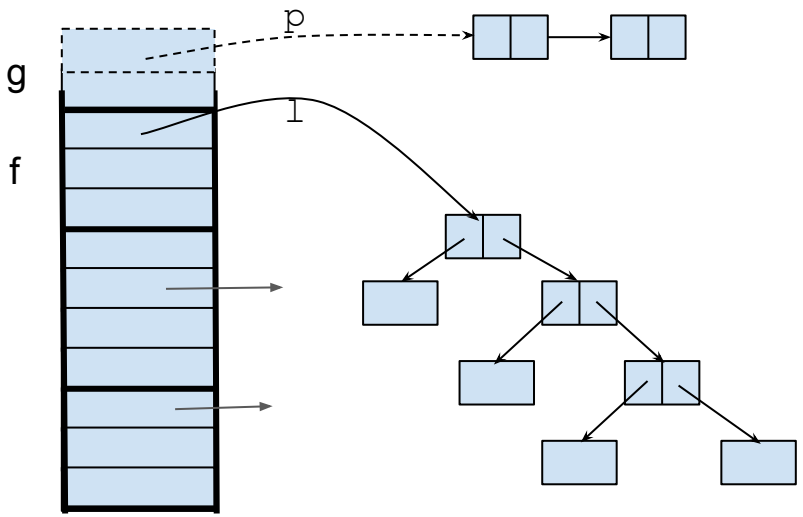
Python has Automated Garbage Collection

- Memory is allocated in two places? Stack and Heap
- Memory allocation in Python

```
void f (int i, int j)
 {
   lptr l;
   create a list and assign to l;
   ...
   g(i)
   ...
   runs out of memory. The Python runtime
   triggers a garbage collection
}

void g (int k)
{
  lptr p;
  create a list and assign to p;
  use p;

  programmer does not have to free p}
```



Python supports lists.

- *Lists and tuples are arguably Python's most versatile, useful data types. You will find them in virtually every nontrivial Python program.*

From: https://realpython.com/python-lists-tuples/

  - Lists are ordered.
  - Lists can contain any arbitrary objects.
  - List elements can be accessed by index.
  - Lists can be nested to arbitrary depth.
  - Lists are mutable.
  - Lists are dynamic.

Python supports lists.

- Lists are ordered.

```
>>> [1,2,3,4] == [4,1,3,2]
False
```

- Lists are heterogenous (the same list can have different types of objects)

```
>>> a = [21.42, 'foobar', 3, 4, 'bark', False, 3.14159]
>>> mixed = [int, collatz, sin, fact]
>>> empty = []
```

Python supports lists.

- List elements can be accessed by index.

```
          0          1          2  3          4          5          6
>>> a = [21.42, 'foobar', 3, 4, 'bark', False, 3.14159]
         -7         -6         -5 -4         -3         -2         -1
```

```
>>> a[-5]
3
>>> a[2:4] #Slice:from a[2] upto,but not including,a[4]
[3,4]
```

Python supports lists.

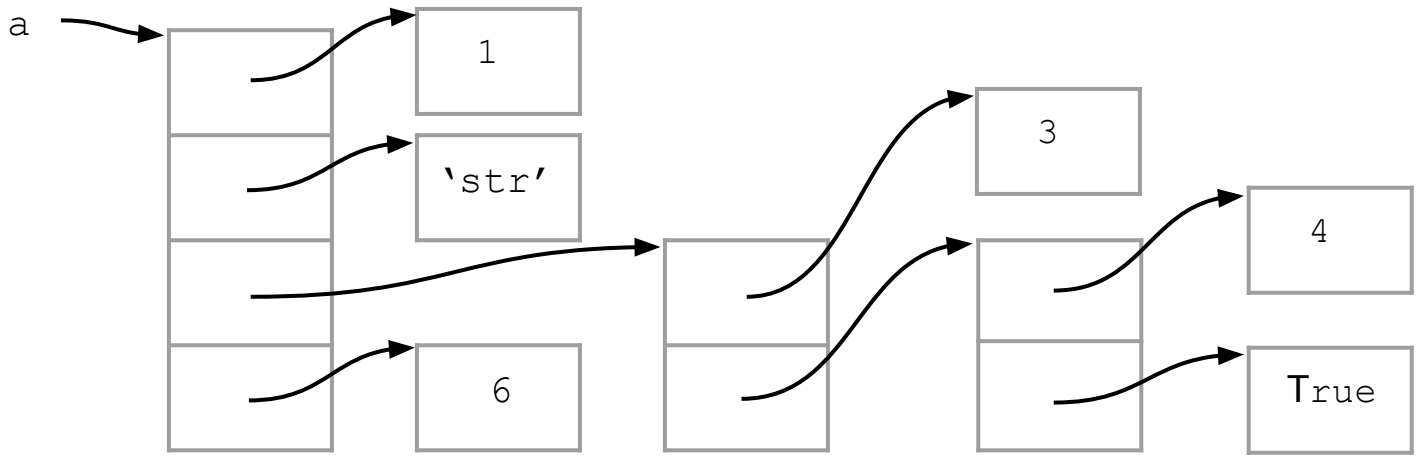
- a[:n] is the same as a[0:n]
- a[n:] is the same as a[n:len(a)]
- Thus a[:n] + a[n:] = a = a[:] (+ is append)
- You can also add a stride or step in a slice

```
>>> a = ['foo', 'bar', 'baz', 'bark', 'qux', 'cor']
>>> a[0:5:2]
['foo', 'baz', 'qux']
>>> a[5:0]
[]
>>> a[5:0:-1]
['cor', 'qux', 'bark', 'baz', 'bar']
```

Lists representation

- Lists can be nested to arbitrary depth

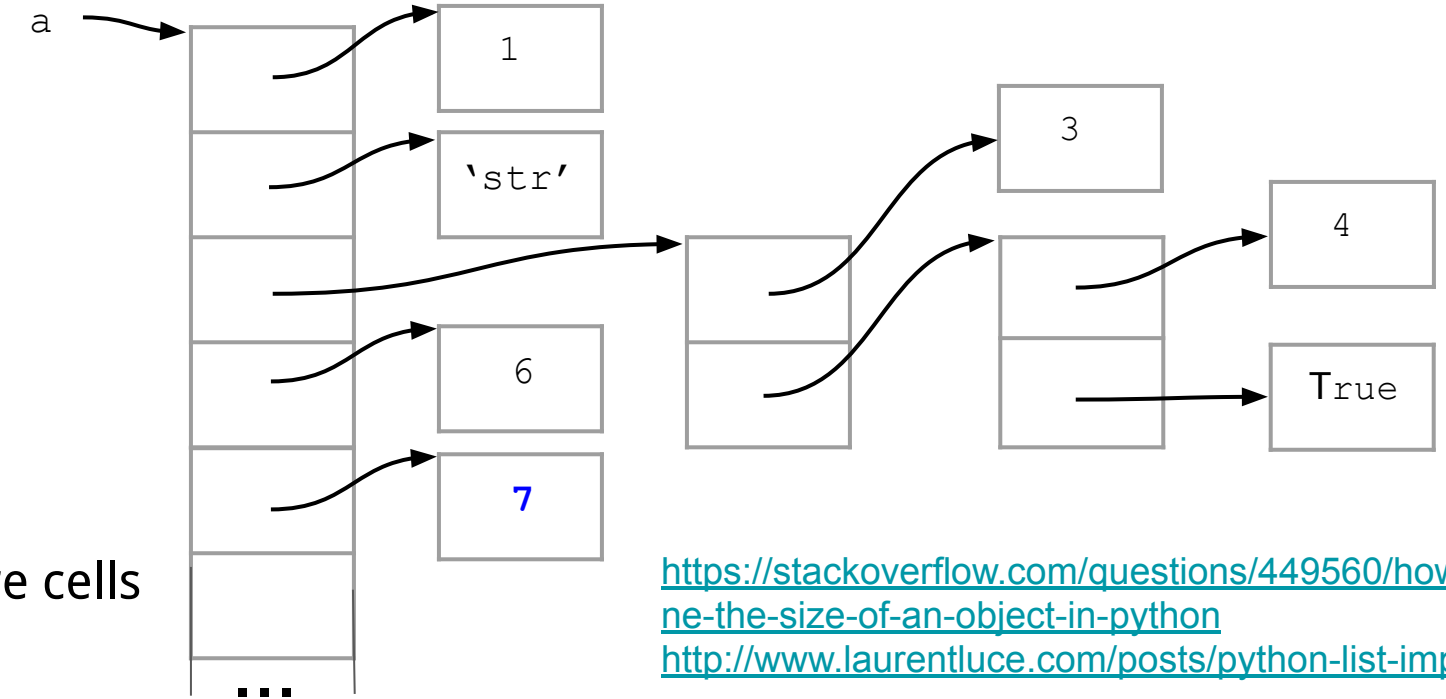
```
>>> a = [1, 'str', [3, [4, True]], 6]
```



List representation

- Lists are represented as dynamic arrays with the varying allocation sizes.

```
>>> a = [1, 'str', [3, [4, True]], 6, 7]
```



4 more cells

https://stackoverflow.com/questions/449560/how-do-i-determine-the-size-of-an-object-in-python

http://www.laurentluce.com/posts/python-list-implementation/

Lists are mutable

```
>>> a.insert(2,'name')
>>> a
[1, 'str','name', [3, [4, True]], 6]
```

- Different from the behavior some functional languages which don't allow mutations.

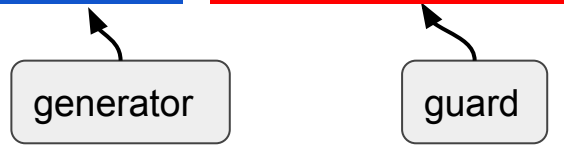
Read how lists operations are done in such languages. Read the section titled "Examples of persistent data structures" : https://en.wikipedia.org/wiki/Persistent_data_structure.

How would you perform an insert in such a language?

List comprehension

- Python has a powerful feature called list comprehension.
- Example:

```
>>> [x*x for x in [1,2,3,4,5,6,7] if x % 2 == 0]
[4, 16, 36]
```



- General form:

`[expr qual1 qual2 qual3 qual4...]`, each `qual` is a generator or a `guard`

- `Example:`

```
>>> [x+y for x in [1,2,3] for y in [5, 6, 7]]
[6, 7, 8, 7, 8, 9, 8, 9, 10][
```

List comprehension

- Here is quicksort using list comprehension

```
def qsort(list):
    if list == []:
        return []
    pivot = list[0]
    l = qsort([x for x in list[1:] if x < pivot])
    u = qsort([x for x in list[1:] if x >= pivot])
    return (l + [pivot] + u)
```

Dictionaries (aka maps, hash-tables)

- Stores objects identified by keys


```
phonebook = {"bob": 7387, "alice": 3719, "jack": 7052,}
```

- Dictionary comprehensions work like list comprehensions.

```
phonebook = {name:number for name in ["bob", "alice", "jack"]
                         for number in [7387, 3719, 7052]}
```

- Access using a syntax similar to lists or arrays: `phonebook["bob"]`
- The key object should be hashable. More about dictionaries here.

Other data structures

- Tuples are immutable containers:
  - `point_3d = (4.6, 5.7, -2.1)`
- `array.array`: The familiar C-like array with all elements having the same type.
  - `arr = array.array("f", (1.0, 1.5, 2.0, 2.5))`
  - These are mutable, dynamic and homogenous
  - In contrast, lists are mutable dynamic and heterogeneous.
  - More time and space efficient. Why?
- sets are unordered collections
  - `vowels = {"a", "e", "i", "o", "u"}`
  - `There is also a variant called multiset`
- `For a deep dive into python data structures, visit the links to the left of this page.`
- `Interested in time complexity of operations on various python data structures? Visit this page.`

Functions

In Python, functions can be treated as any other values

```
def f (x, y):
    return x**2 + y
```

- We can assign the function `f` to a variable.

```
>>> g = f
>>> g(3,2)
```

- We can talk about a function without naming it.

```
>>> (lambda x, y: x ** 2 + y)(3,2)
```

However the notion of lambda is very restricted in Python.

- We can pass functions to other functions:

```
def f (x,y):
    return x**2
>>> map(f, [1,2,3])
[1,4,9]
```

Classes and objects

```
class Account ():

    init_message = "Welcome customer"    #class variable

    def __init__(self, account_holder, balance):  #constructor
        self.account_holder = account_holder      #instance variable
        self.balance = balance

    def show(self):                                #class method
        print(self.account_holder)
        print(self.balance)

    def withdraw (self, amount):
        if  amount <= self.balance:
            self.balance = self.balance - amount
            return self.balance
        else:
            return "Insufficient funds"

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

```
>>> my_account = Account("Amitabha
Sanyal", 1000)
>>> my_account.show()
Amitabha Sanyal
1000
>>> print(my_account.init_message)
Welcome customer
```

Classes and objects

```
class Protected_Account (Account):                          #Derived and base classes

  """Modelling a password protected  bank account"""  #Documentation

   def __init__(self, passwd, account_holder, balance): #constructor of derived
     super().__init__(account_holder, balance)           #constructor of base
     self.__password = passwd                            #private member
                                                          #Note: show() not defined
   def withdraw (self, amount, passwd):                  #withdraw modified
     if passwd == self.__password:
        return super().withdraw(amount)                  #withdraw of base
     else:
        return "Transaction failed, wrong password"

   def deposit(self, amount):
       return super().deposit(amount)
```

```
>>> pacc = Protected_Account ("lksdj", "Sanyal", 1000)
>>> pacc.show()
>>> pacc.withdraw(30,"lksd")
>>> print(pacc.__password)
```

Example 1:

that following input represents the scores of players in IPL matches:

```
3                          # number of following lines that follow
match1:p1-9,p2-38          # match1 details, p and P are players
match2:p3-19,P1-49
m3:p3-1,p4-6,p1-91         # A match can also be called m
```

The output is:

```
{'match1':{'p1':9, 'p2':38},
 'match2':{'p3':19, 'P1':49},
 'm3':{'p3':1, 'p4':6, 'p1':91}}                    # A dictionary
[('p1', 100), ('P1', 49), ('p2', 38), ('p3', 20), ('p4', 6)]    # and a list sorted by aggregate runs.
```

Example 1:

We can break the problem into:

- Read the line of input.
- Read each of the subsequent lines and form the dictionary
- Form the aggregate the create the list

1. Read input: the basic command is `input()`

```
>>> no = int(input())
23                      # input using the keyboard
>>> no
23
```

`input()` reads from the keyboard as a string. It has to be cast to the actual type.
Read more variations of `input()` from:
https://realpython.com/python-input-output/

Example 1:

2. Read each of the subsequent lines and form the dictionary

```
d = {}                              # initialize dictionary d
for i in range(no):                 # For all subsequent lines
    str1 = input()                  # Read input
    l1 = str1.split(':')            # Split the input on character ':'
    d[l1[0]] = {}                   # Initialize dictionary for match
    l2 = l1[1].split(',')           # Split player details
    for j in range(len(l2)):        # For each player
        l3 = l2[j].split('-') # Separate name of player and runs
        d[l1[0]][l3[0]] = int(l3[1]) # Fill inner dictionary
                                     # l1[0]- match name
                                     # l3[0]- player name
                                     # int(l3[1])-runs scored
```

Example 1:

2. Read each of the subsequent lines and form the dictionary

```
d1 = {}                              # d1::{players, aggregate runs}
for i in d.keys():                   # For every match
    for j in d[i].keys():            # For every player in match
        try:
            d1[j] += d[i][j]         # If player already has entry, add
        except:
            d1[j] = d[i][j]          # else create an entry in d1

s = sorted(d1.items(), key=lambda kv:(kv[1],kv[0]),reverse=True)
                                     # reverse sort into a list,
                                     # First sort by runs, then player

print(d)
print(s)
```

Dictionary operations.

- Dictionary operations that you ought to be familiar with:
  - `d.clear()` -- Clears a dictionary
  - `d.get(<key>[, <default>])` -- Returns the value associated with key.
  - `d.items()` -- Returns a list of key-value pairs in a dictionary.
  - `d.keys()` -- Returns a list of keys in a dictionary.
  - `d.values()` -- Obvious
  - `d.pop(<key>[, <default>])` -- Removes a key from a dictionary, if it is present, and returns its value.
  - `d.update(<obj>)` -- Merges a dictionary with another dictionary/list.

This link has more information about dictionaries

Example 2

You are the head of an intelligence unit and you suspect one of your staff to be a spy. The only clue that you have is the suspect's diary, called `MyDiary.txt`, which contains amongst a rambling text some email addresses and phone numbers including your own. Your hypothesis is that the suspect is a spy if he has more conversations with some contact other than yourself.

Write a Python program to first print your own occurrence frequency on a line:

```
my frequency: <frequency>
```

The program must report all the frequent contacts in the format below

```
Spy alert! <spy's contact> <frequency>
Spy alert! …
```

Otherwise conclude with:

```
Alls well, no spy!!!
```

Python supports lists.

Your program will be invoked like:

```
python3 spy.py <path-to-diary> <mycontact>
# mycontact could be an email or phone no.
```

Sample output:

```
my frequency: 7
Cheater alert! emokid@niceguys.com 10
```

Example 2

- Here is a semi-formal description of email id

> *<email id>* = *<local part>* **@** *<domain>*
> *<local part>* = one or more *<alphanumeric>*s separated by *<dot_or_us>*
> *<alphanumeric>* = one or more letters [a-zA-Z] or digits [0-9]
> *<dot_or_us>* = The character **.** or the character _
> *<domain>* = one or more *<alphanumeric>*s separated by *<dot>*. The last *<alphanumeric>* should be a *<alphabetic>*
> *<alphabetic>* = one or more letters.
> *<phone_no>* = 10 consecutive digits not starting with 0

Sample email id:- fxps_ho.4@anhthu.org

Note: Email and Number will be at a word boundary but may be surrounded or adjacent to punctuations.

- You have to use regular expressions to find the email IDs and phone numbers.

Example 2.

- First import some libraries:

```
import sys
import re
import string
from collections import Counter
```

- Give names to regular expressions for email addresses and phone numbers. Note that re_email has three “groups” and re_number has one.

```
re_email = r'(\b[A-Za-z0-9]+([._][A-Za-z0-9]+)*@([A-Za-z0-9]+[.])+
           [A-Za-z]+\b)'
re_number = r'(\b[1-9][0-9]{9}\b)'  #\b…\b for matching at word
                                    # boundaries only
```

Example 2.

- A `findall` based on re_email will return a list of triples, one corresponding whereas a `findall` based on numbers will result in just a list of numbers.

```
>>> re.findall(re_email, 'as@cse.iitb.ac.in, amit23358@gmail.com')
[('as@cse.iitb.ac.in','','ac.'),('amit23358@gmail.com','','gmail.')]

>>> re.findall(re_number, '1234567890, 9999999999')
['1234567890', '9999999999']
```

- We use list comprehension to extract the first element (the match of the outer grouping):

```
emails = []; numbers = []
fp = open(sys.argv[1],"r")
for line in fp.readlines():  #For each line in MyDiary.txt do
    emails_in_line = [x[0] for x in re.findall(re_email,line)]
    emails += emails_in_line;  # Accumulate all emails on the line
    numbers += re.findall(re_number,line)  # Same for phone nos.
```

Example 2.

- Here is an outline of the rest of the code:

```
count_emails = Counter(emails) # Creates a dict {e_mail, count}
count_numbers = Counter(numbers)
isNumber = sys.argv[2].isdigit()  # Is my contact phone or email
spy_dict = {} # Initialize a dictionary of spies
isSpy = False # To track whether a spy has been found or not
if isNumber:
    print('my frequency:',  count_numbers[sys.argv[2]])
    for no in list(set(numbers)): # Makes the numbers unique
        if count_numbers[no] > count_numbers[sys.argv[2]]:
            isSpy = True
            spy_dict[no] = count_numbers[no]
else: ...      # Do the same for emails
```

- Now it is a simple matter to write the rest of the program.