# Installing Python

- Python is available on all platforms: Linux, MacOS and Windows
- Two main flavours of Python
  - Python 2.7
  - Python 3 (currently 3.7.x)
- We will work with Python 3

# Python interpreter

- Python is basically an interpreted language
  - Load the Python interpreter
  - Send Python commands to the interpreter to be executed
  - Easy to interactively explore language features
  - Can load complex programs from files
    - `>>> from filename import *`

# A typical Python program

```
def function_1(..,..):
  …
def function_2(..,..):
  …
     ⋮
def function_k(..,..):
  …

statement_1
statement_2
   ⋮
statement_n
```

- Interpreter executes statements from top to bottom
- Function definitions are “digested” for future use
- Actual computation starts from `statement_1`

# A more messy program

```
statement_1

def function_1(..,..):
  …

statement_2
statement_3

def function_2(..,..):
  …

statement_4
  ⋮
```

- Python allows free mixing of function definitions and statements
- But programs written like this are likely to be harder to understand and debug

# Assignment statement

- Assign a value to a name

```
i = 5
j = 2*i
j = j + 5
```

- Left hand side is a name
- Right hand side is an expression
  - Operations in expression depend on type of value

# Numeric values

- Numbers come in two flavours
  - `int` — integers
  - `float` — fractional numbers
- `178, -3, 4283829` are values of type `int`
- `37.82, -0.01, 28.7998` are values of type `float`

# Operations on numbers

- Normal arithmetic operations: `+,-,*,/`
  - Note that / always produces a float
  - `7/3.5` is `2.0`, `7/2` is `3.5`
- Quotient and remainder: `//` and `%`
  - `9//5` is `1`, `9%5` is `4`
- Exponentiation: `**`
  - `3**4` is `81`

# Other operations on numbers

- `log(), sqrt(), sin(), …`
- Built in to Python, but not available by default
- Must include `math` “library”
    - `from math import *`

# Boolean values: `bool`

- `True, False`
- Logical operators: `not, and, or`
    - `not True` is `False`, `not False` is `True`
    - `x and y` is `True` if both of `x,y` are `True`
    - `x or y` is `True` if at least one of `x,y` is `True`

# Comparisons

- `x == y, a != b,`
  `z < 17*5, n > m,`
  `i <= j+k, 19 >= 44*d`
- Combine using logical operators
  - `n > 0 and m%n == 0`
- Assign a boolean expression to a name
  - `divisor = (m%n == 0)`

# Examples

```
def divides(m,n):
  if n%m == 0:
    return(True)
  else:
    return(False)

def even(n):
  return(divides(2,n))

def odd(n):
  return(not even(n))
```

# Strings — type `str`

- Type string, `str`, a sequence of characters
  - A single character is a string of length 1
  - No separate type `char`
- Enclose in quotes — single, double, even triple!

```
city = 'Chennai'

title = "Hitchhiker's Guide to the Galaxy"

dialogue = '''He said his favourite book is
"Hitchhiker's Guide to the Galaxy”'''
```

# Strings as sequences

- String: sequence or list of characters
- Positions 0,1,2,...,n-1 for a string of length n
  - `s = "hello"`

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| h | e | l | l | o |
| -5 | -4 | -3 | -2 | -1 |

- Positions -1,-2,... count backwards from end
  - `s[1] == "e", s[-2] = "l"`

# Operations on strings

- Combine two strings: concatenation, operator `+`
  - `s = "hello"`
  - `t = s + ", there"`
  - `t` is now `"hello, there"`
- `len(s)` returns length of `s`

# Names, values and types

- Types in Python are dynamic, but strong
- Values have types
  - Type determines what operations are legal
- Names inherit their type from their current value
  - Type of a name is not fixed
  - Unlike languages like C, C++, Java where each name is “declared” in advance with its type

# Names, values and types

- Names can be assigned values of different types as the program evolves

```
i = 5    # i is int
i = 7*1  # i is still int
j = i/3  # j is float, / creates float
…
i = 2*j  # i is now float
```

- `type(e)` returns type of expression `e`

# Extracting substrings

A slice is a “segment” of a string

- `s = "hello"`
- `s[1:4]` is `"ell"`
- `s[i:j]` starts at `s[i]` and ends at `s[j-1]`
- `s[:j]` starts at `s[0]`, so `s[0:j]`
- `s[i:]` ends at `s[len(s)-1]`, so `s[i:len(s)]`

# Modifying strings

- Cannot update a string “in place”
  - `s = "hello",` want to change to `"help!"`
  - `s[3] = "p"` — error!
- Instead, use slices and concatenation
  - `s = s[0:3] + "p!"`
- Strings are immutable values (more later)

# Lists

- Sequences of values

  `factors = [1,2,5,10]`

  `names = ["Anand","Charles","Muqsit"]`

- Type need not be uniform

  `mixed = [3, True, "Yellow"]`

- Extract values by position, slice, like `str`

  `factors[3]` is `10`, `mixed[0:2]` is `[3,True]`

- Length is given by `len()`

  `len(names)` is `3`

# Nested lists

- Lists can contain other lists

```
nested = [[2,[37]],4,["hello"]]
```

`nested[0]` is `[2,[37]]`

`nested[1]` is `4`

`nested[2][0][3]` is `"l"`

`nested[0][1:2]` is `[[37]]`

# Updating lists

- Unlike strings, lists can be updated in place

  `nested = [[2,[37]],4,["hello"]]`

  `nested[1] = 7`

  `nested` is now `[[2,[37]],7,["hello"]]`

  `nested[0][1][0] = 19`

  `nested` is now `[[2,[19]],7,["hello"]]`

- Lists are mutable, unlike strings

# Mutable vs immutable

- What happens when we assign names?

```
x = 5
y = x
x = 7
```

- Has the value of `y` changed?
    - No, why should it?
    - Does assignment copy the value or make both names point to the same value?

# Mutable vs immutable ...

- Does assignment copy the value or make both names point to the same value?
- For immutable values, we can assume that assignment makes a fresh copy of a value
  - Values of type `int, float, bool, str` are immutable
- Updating one value does not affect the copy

# Mutable vs immutable …

- For mutable values, assignment does not make a fresh copy

```
list1 = [1,3,5,7]
list2 = list1
list1[2] = 4
```

- What is `list2[2]` now?
  - `list2[2]` is also `4`
- `list1` and `list2` are two names for the same list

# Copying lists

- How can we make a copy of a list?
- A slice creates a new (sub)list from an old one
- Recall `l[:k]` is `l[0:k]`, `l[k:]` is `l[k:len(l)]`
- Omitting both end points gives a full slice

  `l[:] == l[0:len(l)]`

- To make a copy of a list use a full slice

  `list2 = list1[:]`

# Tuples

- Simultaneous assignments

```
(age,name,primes) = (23,"Kamal",[2,3,5])
```

  - One line swap! `(x,y) = (y,x)`

- Assign a tuple of values to a name

```
point = (3.5,4.8)
```

- Extract positions, slices:

```
ycoordinate = point[0]
```

- Tuples are immutable: `point[1] = 8.7` is an error

# Control flow

* Need to vary computation steps as values change
* Control flow — determines order in which statements are executed
    * Conditional execution
    * Repeated execution — loops
    * Function definitions

# Conditional execution

```
if m%n != 0:
  (m,n) = (n,m%n)
```

- Second statement is executed only if the condition `m%n != 0` is `True`
- Indentation demarcates **body** of `if` — must be uniform

```
if condition:
  statement_1  # Execute conditionally
  statement_2  # Execute conditionally
statement_3    # Execute unconditionally
```

# Alternative execution

```
if m%n != 0:
   (m,n) = (n,m%n)
else:
   gcd = n
```

- `else:` is optional

# Shortcuts for conditions

- Numeric value `0` is treated as `False`
- Empty sequence `""`, `[]` is treated as `False`
- Everything else is `True`

```
if m%n:
  (m,n) = (n,m%n)
else:
  gcd = n
```

# Multiway branching, `elif:`

```
if x == 1:
  y = f1(x)
else:
  if x == 2:
    y = f2(x)
  else:
    if x == 3:
      y = f3(x)
    else:
      y = f4(x)
```

```
if x == 1:
  y = f1(x)
elif x == 2:
  y = f2(x)
elif x == 3:
  y = f3(x)
else:
  y = f4(x)
```

# Loops: repeated actions

- Repeat something a fixed number of times

```
for i in [1,2,3,4]:
    y = y*i
    z = z+1
```

- Again, indentation to mark body of loop

# Repeating n times

- Often we want to do something exactly `n` times

```
for i in [1,2,..,n]:
   . . .
```

- `range(0,n)` generates sequence `0,1,…,n-1`

```
for i in range(0,n):
   . . .
```

- `range(i,j)` generates sequence `i,i+1,…,j-1`
  - More details about `range()` later

# Example

- Find all factors of a number `n`
- Factors must lie between `1` and `n`

```
def factors(n):
  flist = []
  for i in range(1,n+1):
    if n%i == 0:
      flist = flist + [i]
  return(flist)
```

# Loop based on a condition

- If we don't know number of repetitions in advance

```
while condition:
    . . .
```

- Execute body if `condition` evaluates to `True`
- After each iteration, check `condition` again
- Body must ensure progress towards termination!

# Example

- Euclid's gcd algorithm using remainder
- Update `m`, `n` till we find `n` to be a divisor of `m`

```
def gcd(m,n):
  if m < n:
    (m,n) = (n,m)
  while m%n != 0:
    (m,n) = (n,m%n)
  return(n)
```

# A typical Python program

```
def function_1(..,..):
  …
def function_2(..,..):
  …
      ⋮
def function_k(..,..):
  …

statement_1
statement_2
    ⋮
statement_n
```

- Interpreter executes statements from top to bottom
- Function definitions are “digested” for future use
- Actual computation starts from `statement_1`

# Function definition

```
def f(a,b,c):
  statement_1
  statement_2
  ..
  return(v)
  ..
```

- Function name, arguments/parameters
- Body is indented
- `return()` statement exits and returns a value

# Passing values to functions

- Argument value is substituted for name

```
def power(x,n):
  ans = 1
  for i in range(0,n):
    ans = ans*x
  return(ans)
```

```
power(3,5)

x = 3
n = 5
ans = 1
for i in range..
```

- Like an implicit assignment statement

# Passing values …

- Same rules apply for mutable, immutable values
  - Immutable value will not be affected at calling point
  - Mutable values will be affected

# Example

```
def update(l,i,v):
  if i >= 0 and i < len(l):
    l[i] = v
    return(True)
  else:
    v = v+1
    return(False)
```

```
ns = [3,11,12]
z = 8
update(ns,2,z)
update(ns,4,z)
```

- `ns` is `[3,11,8]`
- `z` remains `8`

- Return value may be ignored
- If there is no `return()`, function ends when last statement is reached

# Can pass functions

- Apply `f` to `x` `n` times

```
def apply(f,x,n):
  res = x
  for i in range(n):
    res = f(res)
  return(res)
```

```
def square(x):
  return(x*x)

apply(square,5,2)

square(square(5))

625
```

# Scope of names

- Names within a function have local scope

```
def stupid(x):
  n = 17
  return(x)

n = 7
v = stupid(28)
# What is n now?
```

- `n` is still `7`
  - Name `n` inside function is separate from `n` outside

# Defining functions

- A function must be defined before it is invoked

- This is OK

```
def f(x):
  return(g(x+1))

def g(y):
  return(y+3)

z = f(77)
```

- This is not

```
def f(x):
  return(g(x+1))

z = f(77)

def g(y):
  return(y+3)
```

# Recursive functions

- A function can call itself — recursion

```
def factorial(n):
  if n <= 0:
    return(1)
  else:
    val = n * factorial(n-1)
    return(val)
```

# Some examples

- Find all factors of a number `n`
- Factors must lie between `1` and `n`

```
def factors(n):
  factorlist = []
  for i in range(1,n+1):
    if n%i == 0:
      factorlist = factorlist + [i]
  return(factorlist)
```

# Primes

- Prime number — only factors are 1 and itself
- `factors(17)` is `[1,17]`
- `factors(18)` is `[1,2,3,6,9,18]`

```
def isprime(n):
  return(factors(n) == [1,n])
```

- `1` should not be reported as a prime
  - `factors(1)` is `[1]`, not `[1,1]`

# Primes upto n

- List all primes below a given number

```
def primesupto(n):
  primelist = []
  for i in range(1,n+1):
    if isprime(i):
      primelist = primelist + [i]
  return(primelist)
```

# First n primes

- List the first n primes

```
def nprimes(n):
  (count,i,plist) = (0,1,[])
  while(count < n):
    if isprime(i):
      (count,plist) = (count+1,plist+[i])
    i = i+1
  return(plist)
```

# More about `range()`

- `range(i,j)` produces the sequence `i,i+1,…,j-1`
- `range(j)` automatically starts from `0`; `0,1,…,j-1`
- `range(i,j,k)` increments by `k`; `i,i+k,…,i+nk`
  - Stops with `n` such that `i+nk < j <= i+(n+1)k`
- Count down? Make `k` negative!
  - `range(i,j,-1)`, `i > j`, produces `i,i-1,…,j+1`

# `range()` and lists

- Compare the following
  - `for i in [0,1,2,3,4,5,6,7,8,9]:`
  - `for i in range(0,10):`
- Is `range(0,10) == [0,1,2,3,4,5,6,7,8,9]`?
  - In Python2, yes
  - In Python3, no!

# `range()` and lists

- Can convert `range()` to a list using `list()`
  - `list(range(0,5)) == [0,1,2,3,4]`
- Other type conversion functions using type names
  - `str(78) = "78"`
  - `int("321") = 321`
    - But `int("32x")` yields error

# Lists

- Lists are mutable
  - ```
    list1 = [1,3,5,6]
    list2 = list1
    list1[2] = 7
    ```
  - `list1` is now `[1,3,7,6]`
  - So is `list2`
    ```

# Lists

- On the other hand
  - ```
    list1 = [1,3,5,6]
    list2 = list1
    list1 = list1[0:2] + [7] + list1[3:]
    ```
  - `list1` is now `[1,3,7,6]`
  - `list2` remains `[1,3,5,6]`
- Concatenation produces a new list
    ```

# Extending a list

- Adding an element to a list, in place
  - ```
    list1 = [1,3,5,6]
    list2 = list1
    list1.append(12)
    ```
  - `list1` is now `[1,3,5,6,12]`
  - `list2` is also `[1,3,5,6,12]`
    ```

# List functions

- `list1.append(v)` — extend `list1` by a single value `v`
- `list1.extend(list2)` — extend `list1` by a list of values
  - In place equivalent of `list1 = list1 + list2`
- `list1.remove(x)` — removes first occurrence of `x`
  - Error if no copy of `x` exists in `list1`

# A note on syntax

- `list1.append(x)` rather than `append(list1,x)`
    - `list1` is an object
    - `append()` is a function to update the object
    - `x` is an argument to the function

# List membership

- `x in l` returns `True` if value `x` is found in list `l`

```
# Safely remove x from l
if x in l:
  l.remove(x)

# Remove all occurrences of x from l
while x in l:
  l.remove(x)
```

# Other functions

- `l.reverse()` — reverse `l` in place
- `l.sort()` — sort `l` in ascending order
- `l.index(x)` — find leftmost position of `x` in `l`
  - Avoid error by checking if `x in l`
- `l.rindex(x)` — find rightmost position of `x` in `l`
- Many more … see Python documentation!

# Initialising names

- A name cannot be used before it is assigned a value

```
y = x + 1 # Error if x is unassigned
```

- May forget this for lists where update is implicit

```
l.append(v)
```

- Python needs to know that `l` is a list

# Initialising names ...

```
def factors(n):

  for i in range(1,n+1):
    if n%i == 0:
      flist.append(i)

  return(flist)
```

# Initialising names ...

```
def factors(n):

  flist = []

  for i in range(1,n+1):
    if n%i == 0:
      flist.append(i)

  return(flist)
```

# Sequences of values

- Two basic ways of storing a sequence of values
  - Arrays
  - Lists
- What’s the difference?

# Arrays

- Single block of memory, elements of uniform type
  - Typically size of sequence is fixed in advance
- Indexing is fast
  - Access `seq[i]` in constant time for any `i`
  - Compute offset from start of memory block
- Inserting between `seq[i]` and `seq[i+1]` is expensive
- Contraction is expensive

# Lists

- Values scattered in memory
  - Each element points to the next—“linked” list
  - Flexible size
- Follow i links to access `seq[i]`
  - Cost proportional to `i`
- Inserting or deleting an element is easy
  - “Plumbing”

# Operations

- Exchange `seq[i]` and `seq[j]`
  - Constant time in array, linear time in lists
- Delete `seq[i]` or Insert `v` after `seq[i]`
  - Constant time in lists (if we are already at `seq[i]`)
  - Linear time in array
- Algorithms on one data structure may not transfer to another
  - Example: Binary search

# Python lists

- Are built in lists in Python lists or arrays?
- Documentation suggests they are lists
  - Allow efficient expansion, contraction
- However, positional indexing allows us to treat them as arrays
- Numpy package provides real arrays (later)

# Generalizing lists

- `l = [13, 46, 0, 25, 72]`
- View `l` as a function, associating values to positions
    - `l : {0,1,..,4}` ⟶ `integers`
    - `l(0) = 13, l(4) = 72`
- `0,1,..,4` are keys
- `l[0],l[1],..,l[4]` are corresponding values

# Dictionaries

- Allow keys other than `range(0,n)`
- Key could be a string

```
test1["Dhawan"] = 84
test1["Pujara"] = 16
test1["Kohli"] = 200
```

- Python dictionary
  - Any immutable value can be a key
  - Can update dictionaries in place —mutable, like lists

# Dictionaries

- Empty dictionary is `{}`, not `[]`
  - Initialization: `test1 = {}`
  - Note: `test1 = []` is empty list, `test1 = ()` is empty tuple
- Keys can be any immutable values
  - `int, float, bool, string, tuple`
  - But not lists, or dictionaries

# Dictionaries

- Can nest dictionaries

```
score["Test1"]["Dhawan"] = 84
score["Test1"]["Kohli"] = 200
score["Test2"]["Dhawan"] = 27
```

- Directly assign values to a dictionary

```
score = {"Dhawan":84, "Kohli":200}
score = {"Test1":{"Dhawan":84,
  "Kohli":200}, "Test2":{"Dhawan":50}}
```

# Operating on dictionaries

- `d.keys()` returns sequence of keys of dictionary `d`

```
for k in d.keys():
  # Process d[k]
```

- `d.keys()` is not in any predictable order

```
for k in sorted(d.keys()):
  # Process d[k]
```

- `sorted(l)` returns sorted copy of `l`, `l.sort()` sorts `l` in place
- `d.keys()` is not a list —use `list(d.keys())`

# Operating on dictionaries

- Similarly, `d.values()` is sequence of values in `d`

```
total = 0
for s in test1.values():
  total = total + test1
```

- Test for key using `in`, like list membership

```
for n in ["Dhawan","Kohli"]:
  total[n] = 0
  for match in score.keys():
    if n in score[match].keys():
      total[n] = total[n] + score[match][n]
```

# Dictionaries vs lists

- Assigning to an unknown key inserts an entry

```
d = {}
d[0] = 7  # No problem, d == {0:7}
```

- … unlike a list

```
l = []
l[0] = 7  # IndexError!
```

# Reading from the keyboard

- Read a line of input and assign to `userdata`

```
userdata = input()
```

- Display a message prompting the user

```
userdata = input("Enter a number: ")
```

- Input is always a string, convert as required

```
userdata = input("Enter a number: “)
usernum = int(userdata)
```

# Printing to screen

- Print values of names, separated by spaces

```
print(x,y)
print(a,b,c)
```

- Print a message

```
print("Not a number. Try again")
```

- Intersperse message with values of names

```
print("Values are x:", x, "y:", y)
```

# Fine tuning `print()`

- By default, `print( )` appends new line character `'\n'` to whatever is printed
    - Each `print( )` appears on a new line
- Specify what to append with argument `end="…"`

```
print("Continue on the", end=" ")
print("same line", end=".\n")
print("Next line.")
```

Add space, no new line

Add full stop, new line

```
Continue on the same line.↓
Next line.
```

# Fine tuning `print()`

- Items are separated by space by default

```
(x,y) = (7,10)
print("x is",x,"and y is",y,".")

x is 7 and y is 10 .
```

- Specify separator with argument `sep="..."`

```
print("x is ",x," and y is ",y,".", sep="")

x is 7 and y is 10.
```

# Numpy

* Homogenous multidimensional arrays

```
>>> import numpy as np

>>> a =
np.arange(15).reshape(3,5
)

>>> a
array([[ 0, 1, 2, 3, 4],
       [ 5, 6, 7, 8, 9],
       [10,11,12,13,14]])
)
```

```
>>> a.shape
(3, 5)

>>> a.ndim
2

>>> a.dtype.name
'int64'

>>> a.size
15

>>> type(a)
<type 'numpy.ndarray'>
```

# Numpy

- Array creation

```
>>> a =
 np.array([2,3,4])

>>> a
array([2, 3, 4])

>>> a.dtype
dtype('int64')
```

```
>>> b =
  np.array([(1.5,2,3),
            (4,5,6)])

>>> b
array([[ 1.5,  2. ,  3. ],

       [ 4. ,  5. ,  6. ]])

>>> b.dtype
dtype('float64')
```

# Basic operations

```
>>> a = np.array( [20,30,40,50] )

>>> b = np.arange( 4 )

>>> b
array([0, 1, 2, 3])

>>> c = a-b

>>> c
array([20, 29, 38, 47])

>>> b**2
array([0, 1, 4, 9])
```

```
>>> 10*np.sin(a)
array([ 9.12945251,
-9.88031624,
7.4511316 ,
-2.62374854])

>>> a<35
array([ True, True,
False, False])
```

# Slicing

```
>>> a = np.arange(10)**3

>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512,
729])

>>> a[2]
8

>>> a[2:5]
array([ 8, 27, 64])

>>> a[0:5] = -1000

>>> a
array([-1000, -1000, -1000, -1000, -1000,   125,   216,
343,   512,   729])
```

# Iteration

```
>>> for i in a:
...     print(i**(1/3.))

nan
nan
nan
nan
nan
5.0
6.0
7.0
8.0
9.0
```

# Summary

- Python combines simple syntax with rich features
  - Strings, lists, tuples, dictionaries
- Numpy library implements arrays
- Sklearn library implements many ML models
- Deep learning interface to Tensorflow

# Online resources

- https://www.python.org/, Python
- http://www.numpy.org/, NumPy
- http://scikit-learn.org/stable/, scikit-learn
- https://www.tensorflow.org/, TensorFlow