NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 2, Lecture 1

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

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`

`int` vs `float`

- Why are these different types?
- Internally, a value is stored as a finite sequence of 0's and 1's (binary digits, or bits)
- For an `int`, this sequence is read off as a binary number
- For a `float`, this sequence breaks up into a mantissa and exponent
  - Like "scientific" notation: 0.602×10^{24}

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 *`

Names, values and types

- 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`

- Not good style to assign values of mixed types to same name!

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 divides(2,n))
```

Summary

- Values have types
    - Determine what operations are allowed
- Names inherit type from currently assigned value
    - Can assign values of different types to a name
- `int, float, bool`

NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 2, Lecture 2

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

Names, values and types

- Values have types
  - Determine what operations are allowed
- Names inherit type from currently assigned value
  - Can assign values of different types to a name
- `int, float, bool`
- `+,-,*,/,..  and,or,..  ==,!=,>,..`

Manipulating text

- Computation is a lot more than number crunching
- Text processing is increasingly important
  - Document preparation
  - Importing/exporting spreadsheet data
  - Matching search queries to content

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`
- Will see other functions to manipulate strings later

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)

Summary

- Text values — type `str`, sequence of characters
  - Single character is string of length 1
- Extract individual characters by position
- Slices extract substrings
- `+` glues strings together
- Cannot update strings directly — immutable

NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 2, Lecture 3

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

Types of values in Python

- Numbers: `int, float`
  - Arithmetic operations `+,-,*,/,…`
- Logical values: `bool, {True, False}`
  - Logical operations `not,and,…`
  - Comparisons `==,!=,<,>,<=,>=`
- Strings: `str,` sequences of characters
  - Extract by position `s[i]`, slice `s[i:j]`
  - Concatenation `+`, length `len()`, …

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`

Lists and strings

- For `str`, both a single position and a slice return strings

```
h = "hello"
h[0] == h[0:1] == "h"
```

- For lists, a single position returns a value, a slice returns a list

```
factors = [1,2,5,10]
factors[0] == 1, factors[0:1] == [1]
```

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[:]`

Digression on equality

- Consider the following assignments

```
list1 = [1,3,5,7]
list2 = [1,3,5,7]
list3 = list2
```

- All three lists are equal, but there is a difference
  - `list1` and `list2` are two lists with same value
  - `list2` and `list3` are two names for same list

Digression on equality ...

```
list1 = [1,3,5,7]
list2 = [1,3,5,7]
list3 = list2
```

- `x == y` checks if `x` and `y` have same value
- `x is y` checks if `x` and `y` refer to same object

`list1 == list2` is `True`

`list2 == list3` is `True`

`list2 is list3` is `True`

`list1 is list2` is `False`

Concatenation

- Like strings, lists can be glued together using +

```
list1 = [1,3,5,7]
list2 = [4,5,6,8]
list3 = list1 + list2
```

- `list3` is now `[1,3,5,7,4,5,6,8]`
- Note that `+` always produces a new list

```
list1 = [1,3,5,7]
list2 = list1
list1 = list1 + [9]
```

- `list1` and `list2` no longer point to the same object

Summary

- Lists are sequences of values
  - Values need not be of uniform type
  - Lists may be nested
- Can access value at a position, or a slice
- Lists are mutable, can update in place
  - Assignment does not copy the value
  - Use full slice to make a copy of a list

NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 2, Lecture 4

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

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`

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

- Often 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)
```

Summary

- Normally, statements are executed top to bottom, in sequence
- Can alter the control flow
  - `if … elif … else` — conditional execution
  - `for i in ...` — repeat a fixed number of times
  - `while …` — repeat based on a condition

NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 2, Lecture 5

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

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

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)
```

Summary

- Functions are a good way to organise code in logical chunks
- Passing arguments to a function is like assigning values to names
    - Only mutable values can be updated
- Names in functions have local scope
- Functions must be defined before use
- Recursion — a function can call itself

NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 2, Lecture 6

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

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)
```

`for` and `while`

- `primesupto()`
  - Know we have to scan from `1` to `n`, use `for`
- `nprimes()`
  - Range to scan not known in advance, use `while`

for and while

- Can use `while` to simulate `for`

```
for n in range(i,j):
  statement
```

```
n = i
while n < j:
  statement
  n = n+1
```

```
for n in l:
  statement
```

```
i = 0
while i < len(l):
  n = l[i]
  statement
  i = i+1
```

`for` and `while`

- Can use `while` to simulate `for`
- However, use `for` where it is natural
  - Makes for more readable code
- What makes a good program?
  - Correctness and efficiency — algorithm
  - Readability, ease of maintenance — style
  - What you say, and how you say it