Lecture 1, Part 3 : Introduction to Computing, Problem solving using algorithms

Bijendra Nath Jain <bnjain@iiitd.ac.in> (Section A)

Md. Shad Akhtar <shad.akhtar@iiitd.ac.in> (Section B)

Course outline

- Part 1 Introduction to Computing and Programming (first 2 weeks):
  - Problem solving: Problem statement, algorithm design, programming, testing, debugging
  - Scalar data types: integers, floating point, Boolean, others (letters, colours)
  - Arithmetic, relational, and logical operators, and expressions
  - Data representation of integers, floating point, Boolean
  - Composite data structures: string, tuple, list, dictionary, array
  - Sample operations on string, tuple, list, dictionary, array
  - Algorithms (written in pseudo code) vs. programs
  - Variables and constants (literals): association of names with data objects
  - A language to write pseudo code
  - Programming languages: compiled vs. interpreted programming languages
  - Python as a programming language
  - Computer organization: processor, volatile and non-volatile memory, I/O

Course outline (may change a bit)

- Part 2 Algorithm design and Programming in Python (balance 11 weeks):
  - Arithmetic/Logical/Boolean expressions and their evaluations in Python
  - Input/output statements (pseudo code, and in Python)
  - Assignment statement (pseudo code, and in Python)
  - Conditional statements, with sample applications
  - Iterative statements, with sample applications
  - Function sub-programs, arguments and scope of variables
  - Recursion
  - Modules
  - Specific data structures in Python (string, tuple, list, dictionary, array), with sample applications
  - Searching and sorting through arrays or lists
  - Handling exceptions
  - Classes, and object-oriented programming
  - (Time permitting) numerical methods: Newton Raphson, integration, vectors/matrices operations, continuous-time and discrete-event simulation

Variables and constants or literals

- Literal, or constant:
  - Value of literals does not change (viz. cannot be changed)
  - But one may read (or use/output/print) as many times
  - Kinds of literals : integer, floating point, etc.
    - Integers (or **`int`** in Python): **`-56`**
    - Floating point numbers (or **`float`** in Python): **`-4.5`**
    - **`Boolean`** (also in Python): ‘**`True`**’
    - Strings (**`string`** in Python): ‘**`Hello World`**’
    - Etc.

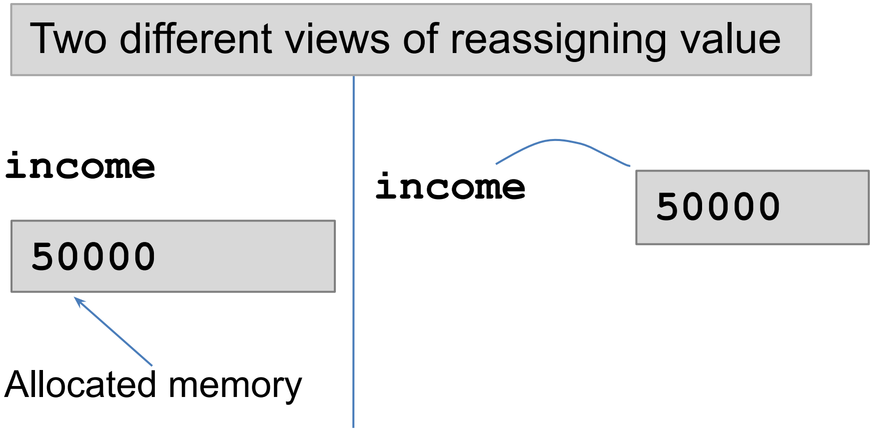
Variables and constants/literals

Variables and constants/literals

- Variables are a way to bind names with objects, or (re)assign value to names
  - Bound values may be changed

```
income = 50000
```
(or income ← 50000)

```
print('income', income)
income = income +1000
print('updated income', income)
```



Output:

```
income 50000
updated income 51000
```

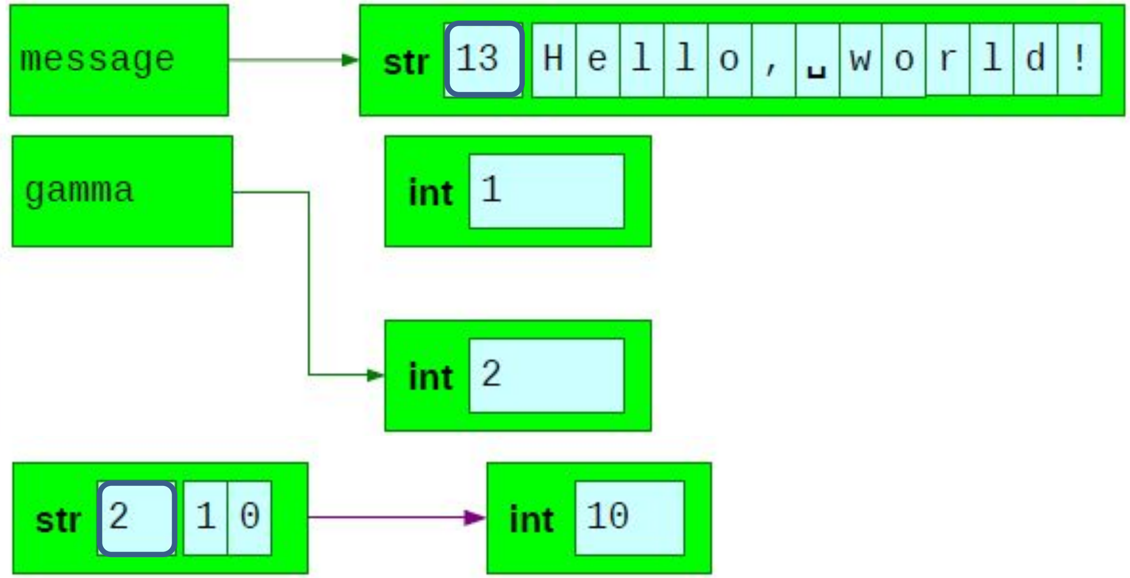
Variables and constants/literals

- How are literals and variables stored in Python, and consequence of binding variables to values:

```
message = ‘Hello, world’

gamma = 1
…
gamma = 2


int(‘10’)
```



- Above **`message, gamma`** are names of two distinct data objects
- In Python:
  - A **name**, e.g. message or gamma, is a string of characters from {A, …, Z, a, …, z, 0, …, 9, _}
  - The first character must be upper or lower case letter
  - Must <u>not</u> be a “reserved word”, e.g. **`int, float, if, function, return,`** etc.

Algorithms vs. programs

- Algorithms (written in pseudo code) vs. programs:
  - Algorithm = a first step towards solving a problem
    - Understand the logic or the method behind the solution
  - Written in a somewhat generic 'pseudo code', with very few constructs
    - Not bogged down by details concerning syntax of a programming language
  - Algorithm uses features available in almost all languages
  - May be analyzed for efficiency and for correctness (but <u>not</u> tested),
  - Algorithm later re-written as a program in a programming language
  - Program in a programming language use special features that may <u>not</u> available in other languages
  - Programs may be tested, debugged and documented
    - One cam document algorithms as well
  - Approach taken in this course:
    - Write an algorithm to solve the problem
    - Analyze it for correctness & efficiency
    - Then re-write it in chosen programming language (e.g. Python), taking advantage of special features that ease the programming task

A language to write pseudo code

- Example algorithm to determine **minT(T1, T2, T3)**

```
input(T1, T2, T3);
minT = T1;
if (T2 < minT) then minT = T2;
if (T3 < minT) then minT = T3;
output(minT)
```

- Above algorithm uses 4 different types of statements:
  - Assignment statement, e.g. **minT = T1**, or **minT = T2**, or **minT = T3**
  - Conditional statement, e.g. **if (T2 < minT) then minT = T2**
  - Input statement, e.g. **input(T1, T2, T3)**
  - Output statement, e.g. **output(minT)**
- Many more statements are required & available to solve more complex problems

Assignment statements

- Assignment statements

  Example of assignment statements

```
vol = height * length * width
tax = 30000 + 0.30*(INCOME-300000)
```

  More generally:

```
v = expr
```

  where **`v`** is variable, and

  `expr` is arithmetic, relational or logical expression for example:

```
30000 + 0.30*(INCOME-300000)
INC > 200000
(marks > 49.99) and (marks < 60)
```

  Note: **`expr`** is first evaluated and then resulting value assigned or bound to **`v`**

Role of ‘;’

- Role of ‘;’ is to separate two statements
  For example:

```
input(T1, T2, T3);
minT = T1;
if (T2 < minT) then minT = T2;
if (T3 < minT) then minT = T3;
output(minT)
```

Python uses different mechanisms to separate two statements, e.g. Write statements is on separate lines

- Or, equivalently

```
input(T1, T2, T3); minT = T1;
if (T2 < minT) then minT = T2;
if (T3 < minT) then minT = T3;
output(minT)
```

- You may even skip the ‘;’ altogether, PROVIDED it is obvious that the two statements are distinct
  E.g.

```
input(T1, T2, T3); minT = T1
if (T2 < minT) then minT = T2
if (T3 < minT) then minT = T3
output(minT)
```

Conditional statements

- Conditional statements

  Examples:

  <u>if</u> (T2 < T) <u>then</u> T = T2

  <u>if</u> (INC > 100000) and (INC < 200001) <u>then</u> tax = 0.10 * (INC-100000)

  More generally:

  <u>if</u> C1 <u>then</u> S1

  where condition **C1** is Boolean **valued logical expression.** For example:

  INC > 100000) and (INC < 200001)

  (T2 < T)

  S1 is any **statement** (assignment, conditional, input, output , etc.). For example:

  tax = 0.10 * (INC-100000)

  T = T2

  or

  if C2 then S2

Consequently, one may have
<u>if</u> C1 <u>then</u> <u>if</u> C2 <u>then</u> S

Conditional statements

- Two variations of conditional statements:

  `if C1 then S1`

  and

  `if C1 then S1 else S2`

  where **`C1`** is Boolean valued logical expressions, and **`S1, S2`** are two statements
- Examples:

  `if (marks < 30) then grade4credit ='F'`

  `if (marks > 39.99) then grade4audit = 'Pass' else grade4audit = 'Fail'`

Conditional statements

- How about something like this?

  <u>if</u> C1 <u>then</u> <u>if</u> C2 <u>then</u> S1 <u>else</u> S2

  where **C1, C2** are Boolean valued logical expressions, and **S1, S2** are two statements
- Examples:

  <u>if</u> (forAudit=True) <u>then</u> <u>if</u> marks > 39.9 <u>then</u> GR =‘AP’ <u>else</u> GR = ‘AF’

Conditional statements

- Consider what if **`S2`** itself is a conditional statement

  `if C1 then if C2 then S1 else S2`

  where **`C1, C2`** are Boolean valued logical expressions, and **`S1, S2`** are statements
- Examples:

  `if (forAudit=True) then if marks > 39.9 then GR ='AP' else GR = 'AF'`

- Truly, what is the interpretation of:

  `if C1 then if C2 then S1 else S2`

  Is it

  `if C1 then [if C2 then S1 else S2]`

  or

  `if C1 then [if C2 then S1] else S2`

Conditional statements

- Compounding of multiple statements:

  Why do we need this?

  Multiple statements can be “bracketed together” to form ONE statement, or

  A null statement may be bracketed together, or

  Simply force an interpretation

  For example:

```
if x <= 1000 then [n = n + 1; x = 2 * x]
```

  is equivalent to

```
if x <= 1000 then [n = n + 1; x = 2 * x] else [ ]
```

Python uses different mechanisms to indicate that a sequence of statements is one compound statement

Conditional statements

- The intended interpretation of:

```
if C1 then if C2 then S1 else S2
```

can be forced as follows:

```
if C1 then [if C2 then S1 else S2]
```

Or

```
if C1 then [if C2 then S1] else S2
```

- To understand this better consider:

```
if x < 0 then [if y < 0 then output("yes") else output("no")]
```

	Y < 0	Y >= 0
X < 0	yes	no
X >= 0	-	-

or

```
if x < 0 then [if y < 0 then output("yes")] else output("no")
```

	Y < 0	Y >= 0
X < 0	yes	-
X >= 0	no	no

Conditional statements

- Example use of nested if-then-else statement:

INCOME	Tax
Less than 100000	0
Between 100000 & 200000	0 plus 10% of INC excess of 100000
Between 200000 & 300000	10000 plus 20% of INC excess of 200000
Above 300000	30000 plus 30% of INC excess of 300000

Conditional statements

- Example use of nested if-then-else statement:

INC	Tax
Less than 100000	0
Between 100000 & 200000	0 plus 10% of INC excess of 100000
Between 200000 & 300000	10000 plus 20% of INC excess of 200000
Above 300000	30000 plus 30% of INC excess of 300000

```
input(INC);
if INC > 300000
then Tax = 30000 + 0.30*(INC-300000)
else if INC > 200000 ~~and INC ≤ 300000~~
     then Tax = 10000 + 0.20*(INC-200000)
     else if INC > 100000 ~~and INC ≤ 200000~~
          then Tax = 0000 + 0.10*(INC-100000)
          else ~~if INC ≤ 100000 then~~ Tax = 0
```

Conditional statements

- Example use of nested if-then-else statement:

Income, INC	Tax, T
Less than 100000	0
Between 100000 & 200000	0 plus 10% of INC excess of 100000
Between 200000 & 300000	10000 plus 20% of INC excess of 200000
Above 300000	30000 plus 30% of INC excess of 300000

```
input(INC);
case [
INC > 300000: Tax = 30000 + 0.30*(INC-300000)
INC > 200000: Tax = 10000 + 0.20*(INC-200000)
INC > 100000: Tax = 0000 + 0.10*(INC-100000)
True: Tax = 0
]
```

Conditional statements

- Case statement in lieu of nested if-then-else statements

```
if C1 then S1 else [if C2 then S2 else [if C3 then S3 else [if C4
   then S4 else [ ] ]]]

case [
<C1>: S1;
<C2>: S2;
<C3>: S3;
<C4>: S4
]

case [
INC > 300000: Tax = 30000 + 0.30*(INC-300000);
INC > 200000: Tax = 10000 + 0.20*(INC-200000);
INC > 100000: Tax = 0000 + 0.10*(INC-100000);
True:         Tax = 0
]
```

Conditional statements

- Case statement in lieu of nested if-then-else statements

```
If C1 then S1 else [if C2 then S2 else [if C3 then S3
   else [if C4 then S4 else [] ]]
```

```
case [
<C1>: S1;
<C2>: S2;
<C3>: S3;
<C4>: S4
]
```

```
case [
INC > 100000: T = 0000 + 0.10*(INC-100000);
INC > 200000: T = 10000 + 0.20*(INC-200000);
INC > 300000: T = 30000 + 0.30*(INC-300000);
True:         T = 0
]
```

```
case [
INC > 300000: Tax = 30000 + 0.30*(INC-3000
INC > 200000: Tax = 10000 + 0.20*(INC-200000
INC > 100000: Tax = 0000 + 0.10*(INC-1000
True:         Tax = 0
]
```

This algorithm will give completely incorrect results.
Why?

Input/output operations

- Input operation
- Effectively a read followed by an assignment
- Read from keyboard

  Example:

  ```
  input (x1, x2, L, H, W, Name)
  ```
- Output operation

  Effectively a write to computer screen

  Example:

  ```
  output ('The volume is:', H*L*W)
  ```

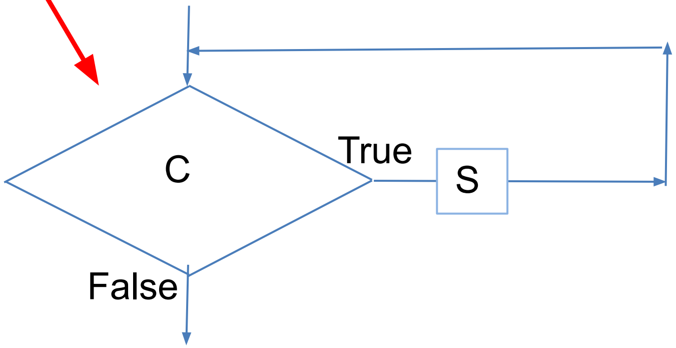
Iteration statements using while-do

- Iteration using **while** `C` **do** `S`

  where **C** is a condition, and **S** is a statement

- **Example:**

```
# Find the largest integer n > 0 such that 2**n < 1000
    n = 1; x = 2**n;
    while x < 1000 do [n = n+1; x = 2**n]
    output('largest n>0 such that 2**n < 1000"', n-1)
```



Iteration statements using while-do

- Iteration using **__while__** **C** **__do__** **S**

  where **C** is a condition, and **S** is a statement

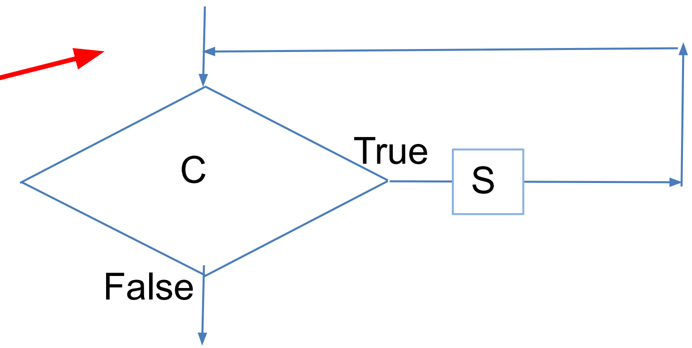
- **Another example:**

```
 # Compute the sum of n numbers
     input(n);
     if n > 0
     then [k = 1; sum = 0;
          while k ≤ n do [input(x); sum = sum + x; k = k +1];
          output('the total is', sum)
          ]
     else output('no numbers to add')
```

Iteration statements using while-do

- Iteration using <u>**`while`**</u> **`C`** <u>**`do`**</u> **`S`**

  or using the <u>**`repeat`**</u> **`S`** <u>**`until`**</u> **`C`**

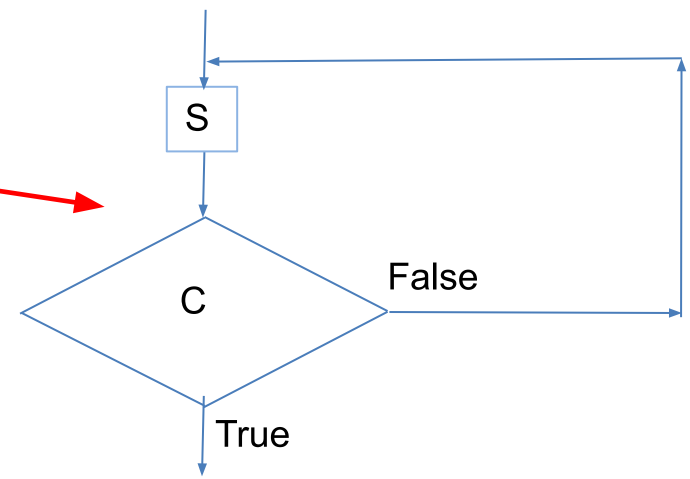


- Note:

  `while C do S`

  is the same as:

  `if C then repeat S until not C`



- And

  `repeat S until C`

  is the same as:

  `S; while not C do S`

Iteration statements using for loop

- Iteration using **for k in <sequence> do S**
- **<sequence>** is an ordered set of objects, typically integers

- Example:

```
# Compute the dot-product of two vectors U and V
# vector U = [u(1), u(2), …, u(n)], n>0
# vector V = [v(1), v(2), …, v(n)], n>0
    input(n);
    for i in <1, 2, …, n> [input(u(i))];
    for j in <1, 2, …, n> [input(v(j))];
    sum = 0;
    for k in <1, 2, …, n> [sum = sum + u(k)*v(k)];
    output('Dot product U*V of vectors U and V is ', sum)
```

Iteration statements using for loop

- Iteration using `for k in <sequence> do S`
- `<sequence>` is an ordered set of objects, typically integers

- Problem1: Given `[t(1), t(2),` …, `t(n)],` find the smallest `t(k)`
- Problem2: Given `[t(1), t(2),` …, `t(n)],` find `k, t(k)= T`

Q&A

- On algorithms
- On Python programs
- On testing
- On debugging
- On documentation
- On scalar data items
- On structured data
- On representation of scalar data