Lecture 1, Part 2: Introduction to Computing –Problem Solving and Data Manipulation

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

Scalar and structured data types

- Computing is about ‘data’, and its manipulation
- Data is of:
  - Scalar data types
  - Structured data types
- Scalar data types:
  - **{‘None’}** only in Python
  - Natural number: **{0, 1, 2, …}**
  - (Signed) integer: **{…, -2, -1, 0, 1, 2, …}**
  - ~~(signed) real numbers~~
  - (Signed) floating point number, includes all positive & negative numbers, but only approximations to rational & irrational numbers:
  - **{…, -2.1, …, ~~-1/3~~, …, 0.00, …, ~~4/3~~, …, 3.1415926535, ~~π~~, …, 4.023, …, ~~$2.1*10^{89}$~~}**
  - Boolean: {‘**False**’, ‘**True**’}
  - Characters: **{…, A, B, …, Z, a, b, …, z, ‘.’, ‘@’, …}**
- Structured data types (will be discussed as we go along):
  - String
  - Tuple
  - List
  - Dictionary
  - Array

Scalar and structured data types

- Computing is about 'data', and its manipulation
- Data is of:
  - Scalar data types
  - Structured data types
- Scalar data types:
  - **{'None'}** only in Python
  - Natural number: **{0, 1, 2, …}**
  - (Signed) integer: **{…, -2, -1, 0, 1, 2, …}**
  - ~~(signed) real numbers~~
  - (Signed) floating point number, includes all positive & negative numbers, but only approximations to rational & irrational numbers:
  - **{…, -2.1, …, ~~-1/3~~, …, 0.00, …, ~~4/3~~, …, 3.1415926535, ~~π~~, …, 4.023, …, ~~$2.1*10^{89}$~~}**
  - Boolean: {'**False**', '**True**'}
  - Characters: **{…, A, B, …, Z, a, b, …, z, '.', '@', …}**
- Structured data types (will be discussed as we go along):
  - String
  - Tuple
  - List
  - Dictionary
  - Array

Scalar and structured data types

- Computing is about 'data', and its manipulation
- Data is of:
  - Scalar data types
  - Structured data types
- Scalar data types:
  - **{'None'}** only in Python
  - Natural number: **{0, 1, 2, …}**
  - (Signed) integer: **{…, -2, -1, 0, 1, 2, …}**
  - ~~(signed) real numbers~~
  - (Signed) floating point number, includes all positive & negative numbers, but only approximations to rational & irrational numbers:
  - **{…, -2.1, …, ~~-1/3~~, …, 0.00, …, ~~4/3~~, …, 3.1415926535, ~~π~~, …, 4.023, …, ~~$2.1*10^{89}$~~}**
  - Boolean: {'**False**', '**True**'}
  - Characters: **{…, A, B, …, Z, a, b, …, z, '.', '@', …}**
- Structured data types (will be discussed as we go along):
  - String
  - Tuple
  - List
  - Dictionary
  - Array

Truly, the best we can do to store fraction 0.1 using 16 bits (e.g.) is to save it as 0.099975586

Scalar and structured data types

- Scalar data types:
  - **{‘None’}** only in Python
  - Natural number: **{0, 1, 2, …}**
  - (Signed) integer: **{…, -2, -1, 0, 1, 2, …}**
  - ~~(signed) real numbers~~
  - (Signed) floating point number, includes all positive & negative numbers, but only approximations to rational & irrational numbers:
  - **{…, -2.1, …, ~~-1/3~~, …, 0.00, …, ~~4/3~~, …, 3.1415926535, ~~π~~, …, 4.023, …, ~~2.1*10~~^{89}}**
  - Boolean: {‘**False**’, ‘**True**’}
  - Characters: **{…, A, B, …, Z, a, b, …, z, ‘.’, ‘@’, …}**
- Structured data types (will be discussed as we go along):
  - String
  - Tuple
  - List
  - Dictionary
  - Array
  - Files

Operations on scalar data types

- Computing is about 'data', and **<u>its manipulation</u>**
- Operations on scalar data
  - Arithmetic operations (<u>`op`</u>)
    - <u>`op`</u> `int` □ `int`, such as `– 7 = -7`
    - `int` <u>`op`</u> `int` □ `int` (there is an exception, though)
    - <u>`op`</u> `float` □ `float`
    - `float` <u>`op`</u> `float` □ `float`
  - Relational operations
    - `int` <u>`op`</u> `int` □ `boolean`, such as `4 ≥ 5`
    - `float` <u>`op`</u> `float` □ `boolean`
  - Logical operations (in the context of Boolean data)
    - <u>`op`</u> `boolean` □`oolean`, such as <u>`not`</u> `P`
    - `boolean` <u>`op`</u> `boolean` □ `boolean`
- And operations that are:
  - Unary, such as `– 7 = -7`, or <u>`not`</u> `P`
  - Binary, such as `4 * -6 = -24`, or `P` <u>`and`</u> `Q`
  - Expressions, such as `6 + 5 * 3 = 21` or (`a ≥ 5 * 3` <u>`and`</u> `a < 5^2`)

<u>`op`</u> is short for 'operator' or operation

Operations on scalar data types

- Computing is about ‘data’, and **its manipulation**
- Operations on scalar data
  - **Arithmetic operations** (op)
    - op `int` □ `int`, such as `– 7 = -7`
    - `int` op `int` □ `int` (there is an exception, though)
    - op `float` □ `float`
    - `float` op `float` □ `float`
  - Relational operations
    - `int` op `int` □ `boolean`, such as `4 ≥ 5`
    - `float` op `float` □ `boolean`
  - Logical operations (in the context of Boolean data)
    - op `boolean` □ `boolean`, such as `not P`
    - `boolean op boolean` □ `boolean`
- And operations that are:
  - Unary, such as `– 7 = -7`, or `not P`
  - Binary, such as `4 * -6 = -24`, or `P and Q`
  - Expressions, such as `6 + 5 * 3 = 21` or (`a ≥ 5 * 3 and a < 5^2`)

op is short for ‘operator’ or operation

What would be the outcome of `int` / `int`

Operations on scalar data types

op is short for 'operator' or operation

- Computing is about 'data', and **its manipulation**
- Operations on scalar data
  - Arithmetic operations (op)
    - op `int` □ `int`, such as `– 7 = -7`
    - `int` op `int` □ `int` (there is an exception, though)
    - op `float` □ `float`
    - `float` op `float` □ `float`
  - **Relational operations**
    - **`int` op `int` □ `boolean`**, such as **`4 ≥ 5`**
    - **`float` op `float` □ `boolean`**
  - **Logical operations (in the context of Boolean data)**
    - **op `boolean` □ `boolean`**, such as **`not P`**
    - **`boolean` op `boolean` □ `boolean`**
- And operations that are:
  - Unary, such as `– 7 = -7`, or `not P`
  - Binary, such as `4 * -6 = -24`, or `P and Q`
  - Expressions, such as `6 + 5 * 3 = 21` or (`a ≥ 5 * 3 and a < 5^2`)

Operations on scalar data types

- Computing is about 'data', and **<u>its manipulation</u>**
- Operations on scalar data
  - Arithmetic operations (<u>op</u>)
    - <u>op</u> `int` `int`, such as `– 7 = -7`
    - `int` <u>op</u> `int` `int` (there is an exception, though)
    - <u>op</u> `float` >>> `float`
    - `float` <u>op</u> `float` >>> `float`
  - Relational operations
    - `int` <u>op</u> `int` >>> `boolean`, such as `4 ≥ 5`
    - `float` <u>op</u> `float` >>> `boolean`
  - Logical operations (in the context of Boolean data)
    - <u>op</u> `boolean` >>> `boolean`, such as <u>`not`</u> `P`
    - `boolean` <u>op</u> `boolean` >>> `boolean`
- **<u>And operations that are</u>**:
  - Unary, such as `– 7 =`
  - Binary, such as `4 * -6 = -24`, or `P` <u>`and`</u> `Q`
  - Expressions, such as

`6 + 5 * 3 = 21 or (a ≥ 5 * 3` <u>`and`</u> `a < 5^2)`

<u>op</u> is short for 'operator' or operation

Operations on integers

- Operations on **integers** (these are called **'int'** in Python)
  - **Arithmetic** operations result in **integer** values:
    - **Unary** operation:
      - ✔ Negate operation,, such as **- 9 = -9**
    - Binary operations:
      - ✔ Add operation, such as **3 + 4 = 7**
      - ✔ Subtract operation, such as **7 – 9 = -2**
      - ✔ Multiply operation, such as **7 * 8 = 56**
      - ✔ Exponentiation, such as **8^2 = 64** (**a**b** in Python, such as **8**2**)
      - ~~✔ Divide operation such as 6 / 8~~ why do you need this – we will discuss this later
      - ✔ '`mod`' operation, **a mod b**, or remainder of **a** when divided by **b**,

        e.g. **70 mod 9 = 7** (**a%b** in Python, such as **70%9 = 7**)

        `mod` operation is useful in computing GCD(a, b)

        Example: GCD(21, 15) = GCD(15, 21 mod 15) = GCD(15, 6), …, = 3

Operations on integers

- Operations on **integers** (these are called **'int'** in Python)
  - **Arithmetic** operations result in **integer** values:
    - Unary operation:
      - ✔ Negate operation,, such as **- 9 = -9**
    - **Binary** operations:
      - ✔ Add operation, such as **3 + 4 = 7**
      - ✔ Subtract operation, such as **7 – 9 = -2**
      - ✔ Multiply operation, such as **7 * 8 = 56**
      - ✔ Exponentiation, such as **8^2 = 64** (**a**b** in Python, such as **8**2**)
      - ✔ Divide operation such as 6 / 8; do we need this – we will discuss this later
      - ✔ '`mod`' operation, **a mod b**, or remainder of **a** when divided by **b**,

        e.g. **70 mod 9 = 7** (**a%b** in Python, such as **70%9 = 7**)

        `mod` operation is useful in computing GCD(a, b)

        Example: GCD(21, 15) = GCD(15, 21 mod 15) = GCD(15, 6), …, = 3

Operations on integers

- **<u>Relational or compare operations result in Boolean values</u>**
  - Binary operations:
    - ✔ `'<'` such as `5 < 7` is **`True`**
    - ✔ `'>'` such as `5 > 7` is **`False`**
    - ✔ `'≤'` such as `5 ≤ 7` is **`True`** ('`<=`' in Python, such as `5 <= 7` is **`True`**)
    - ✔ `'≥'` such as `5 ≥ 7` is **`False`** ('`>=`' in Python, such as `5 >= 7` is **`False`**)
    - ✔ `'='` such as `5 = 7` is **`False`** ('`==`' in Python, such as `5 == 7` is **`False`**)
    - ✔ `'!='`, or "not equal to", such as `5 != 7` is **`True`**
- Interesting property: there is a "total order" on the set of integers
  - That is, for any pair of <u>distinct</u> numbers, n1 and n2, either n1 < n2 or n1 > n2.
  - Equivalently, given a subset of integers, one can always sort them in non-decreasing or non-increasing order

Operations on integers

- Relational or compare operations result in Boolean values
  - Binary operations:
    - ✔ `'<'` such as `5 < 7` is **`True`**
    - ✔ `'>'` such as `5 > 7` is **`False`**
    - ✔ `'≤'` such as `5 ≤ 7` is **`True`** ('`<=`' in Python, such as `5 <= 7` is **`True`**)
    - ✔ `'≥'` such as `5 ≥ 7` is **`False`** ('`>=`' in Python, such as `5 >= 7` is **`False`**)
    - ✔ `'='` such as `5 = 7` is **`False`** ('`==`' in Python, such as `5 == 7` is **`False`**)
    - ✔ `'!='`, or "not equal to", such as `5 != 7` is **`True`**
- Interesting property: there is a "total order" on the set of integers
  - That is, for any pair of <u>distinct</u> numbers, n1 and n2, either n1 < n2 or n1 > n2.
  - Equivalently, given a subset of integers, one can always sort them in non-decreasing or non-increasing order

Operations on floating point numbers

- **Operations on floating point numbers (these are called ‘`float`’ in Python)**
  - Arithmetic operations result in floating point values:
    - Unary operations:
      - ✔ Negate operation, such as `– 9.5 = -9.5`
    - Binary operations:
      - ✔ Add operation, such as `3.01 + 4.02 = 7.03`
      - ✔ Subtract operation, such as `7.00 – 9.03 = -2.03`
      - ✔ Multiply operation, such as `-7.1 * 2.0 = -14.2`
      - ✔ Exponentiation, e.g. `8.5^2  = 72.25` (this op is `a**b` in Python, e.g. `8.5**2`)
      - ✔ Division operation, such as `-9.6 / 3.0 = -3.2`
      - ✔ Division of integers in Python, such as `3/4 = 0.75` (this is performed after converting integers into floating point numbers)

Operations on floating point numbers

- Operations on floating point numbers (these are called '**`float`**' in Python)
  - Arithmetic operations result in floating point values:
    - Unary operations:
      - ✔ Negate operation, such as **`– 9.5 = -9.5`**
    - Binary operations:
      - ✔ Add operation, such as **`3.01 + 4.02 = 7.03`**
      - ✔ Subtract operation, such as **`7.00 – 9.03 = -2.03`**
      - ✔ Multiply operation, such as **`-7.1 * 2.0 = -14.2`**
      - ✔ Exponentiation, e.g. **`8.5^2  = 72.25`** (this op is **`a**b`** in Python, e.g. **`8.5**2`**)
      - ✔ Division operation, such as **`-9.6 / 3.0 = -3.2`**
      - ✔ Division of integers in Python, such as **`3/4 = 0.75`** (this is performed after converting integers into floating point numbers)

Operations on floating point numbers

- **Relational,** or compare operations result in **Boolean** values
  - Binary operations:
    - ✔ `'<'` such as `-5.0 < 7.8` is `True`
    - ✔ `'>'` such as `-5.0 > 7.8` is `False`
    - ✔ `'≤'` such as `-5.0 ≤ 7.8` is `True` (this is '**<=**' in Python, such as `-5.0 ≤ 7.8`)
    - ✔ `'≥'` such as `-5.0 ≥ 7.8` is `False` (this is '**>=**' in Python, such as `-5.0 ≥ 7.8`)
    - ✔ `'='` such as `-5.0 = 7.8` is `False` (this is '**==**' in Python, such as `-5.0 == 7.8`)
    - ✔ `'!='`, or "not equal to", such as `5 != 7 = True`
  - Question: Is `4.0/3.0 == 1.3333333`?

Operations on floating point numbers

- **Relational**, or compare operations result in **Boolean** values
  - Binary operations:
    - ✔ `'<'` such as `-5.0 < 7.8` is `True`
    - ✔ `'>'` such as `-5.0 > 7.8` is `False`
    - ✔ `'≤'` such as `-5.0 ≤ 7.8` is `True` (this is '**<=**' in Python, such as `-5.0 ≤ 7.8`)
    - ✔ `'≥'` such as `-5.0 ≥ 7.8` is `False` (this is '**>=**' in Python, such as `-5.0 ≥ 7.8`)
    - ✔ `'='` such as `-5.0 = 7.8` is `False` (this is '**==**' in Python, such as `-5.0 == 7.8`)
    - ✔ `'!='`, or "not equal to", such as `5 != 7 = True`
  - Question: Is `4.0/3.0 == 1.3333333`?

Operations on floating point numbers

- **Relational**, or compare operations result in **Boolean** values
  - Binary operations:
    - ✔ `'<'` such as `-5.0 < 7.8` is `True`
    - ✔ `'>'` such as `-5.0 > 7.8` is `False`
    - ✔ `'≤'` such as `-5.0 ≤ 7.8` is `True` (this is '<=' in Python, such as `-5.0 ≤ 7.8`)
    - ✔ `'≥'` such as `-5.0 ≥ 7.8` is `False` (this is '>=' in Python, such as `-5.0 ≥ 7.8`)
    - ✔ `'='` such as `-5.0 = 7.8` is `False` (this is '==' in Python, such as `-5.0 == 7.8`)
    - ✔ `'!='`, or "not equal to", such as `5 != 7 = True`
  - Question: Is `4.0/3.0 == 1.3333333`?
    - ✔ **You should be careful while comparing two floating numbers for equality**

Operations on Boolean data

- Operations on **Boolean** data
  - **Logical operations** that result in Boolean values:
    - **Unary**:
      - ✔ `not` , such as `not P`
    - **Binary**:
      - ✔ `and`, such as `P and Q`
      - ✔ `or`, such as `P or Q`
  - The "TRUTH TABLE" below describes the above operations

P	Q	not P	P and Q	P or Q
FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE
TRUE	TRUE	FALSE	TRUE	TRUE

Precedence rules to evaluate expressions

- Evaluating expressions involving scalar data items, and related operations
- **<u>Why are "precedence rules" required?</u>**

  FOUR examples:

  1. Consider `6 + 5 * 3` :

  – depends upon whether add '`+`' or multiply '`*`' is performed first:

  `6 + (5 * 3) = 21` (if `*` is performed first)

  `(6 + 5) * 3 = 33` (if `+` is performed first)

  2. Similarly consider `6 / 2 / 3 = 1` :

  `(6 / 2) / 3 = 1` (if the first '`/`' is performed first)

  `6 / (2 / 3) = 9` (if the second '`/`' is performed first)

  3. Or consider `6 ^ 2 / 2` :

  `(6^2) / 2 = 18` (if '`^`' is performed first)

  `6^(2 / 2) = 6` (if '`/`' is performed first)

  4. Or consider (<u>`not`</u> `True)` <u>`and`</u> `False` :

  (<u>`not`</u> `True)` <u>`and`</u> `False = False` (if <u>`not`</u> is performed first)

  `not (True` <u>`and `</u>`False) = True` (if <u>`and`</u> is performed first)

Precedence rules to evaluate expressions

- Evaluating expressions involving scalar data items, and related operations
- **Why are "precedence rules" required?**

  FOUR examples:

  1. Consider `6 + 5 * 3` :

  – depends upon whether add '`+`' or multiply '`*`' is performed first:

  `6 + (5 * 3) = 21` (if `*` is performed first)

  `(6 + 5) * 3 = 33` (if `+` is performed first)

  2. Similarly consider `6 / 2 / 3 = 1` :

  `(6 / 2) / 3 = 1` (if the first '`/`' is performed first)

  `6 / (2 / 3) = 9` (if the second '`/`' is performed first)

  3. Or consider `6 ^ 2 / 2` :

  `(6^2) / 2 = 18` (if '`^`' is performed first)

  `6^(2 / 2) = 6` (if '`/`' is performed first)

  4. Or consider (`not True) and False` :

  `(not True) and False = False` (if `not` is performed first)

  `not (True and False) = True` (if `and` is performed first)

Precedence rules to evaluate expressions

- Evaluating expressions involving scalar data items, and related operations
- **<u>Why are “precedence rules” required?</u>**

FOUR examples:

1. Consider `6 + 5 * 3` :

– depends upon whether add ‘`+`’ or multiply ‘`*`’ is performed first:

`6 + (5 * 3) = 21` (if `*` is performed first)

`(6 + 5) * 3 = 33` (if `+` is performed first)

2. Similarly consider `6 / 2 / 3 = 1` :

`(6 / 2) / 3 = 1` (if the first ‘`/`’ is performed first)

`6 / (2 / 3) = 9` (if the second ‘`/`’ is performed first)

3. Or consider `6 ^ 2 / 2 :`

`(6^2) / 2 = 18` (if ‘`^`’ is performed first)

`6^(2 / 2) = 6` (if ‘`/`’ is performed first)

4. Or consider `(not True) and False :`

`(not True) and False = False` (if `not` is performed first)

`not (True and False) = True` (if `and` is performed first)

Precedence rules to evaluate expressions

- Evaluating expressions involving scalar data items, and related operations
- **<u>Why are "precedence rules" required?</u>**

  FOUR examples:

  1. Consider `6 + 5 * 3` :

  – depends upon whether add '`+`' or multiply '`*`' is performed first:

  `6 + (5 * 3) = 21` (if `*` is performed first)

  `(6 + 5) * 3 = 33` (if `+` is performed first)

  2. Similarly consider `6 / 2 / 3 = 1` :

  `(6 / 2) / 3 = 1` (if the first '`/`' is performed first)

  `6 / (2 / 3) = 9` (if the second '`/`' is performed first)

  3. Or consider `6 ^ 2 / 2 :`

  `(6^2) / 2 = 18` (if '`^`' is performed first)

  `6^(2 / 2) = 6` (if '`/`' is performed first)

  4. Or consider (**<u>not</u> True) <u>and</u> False :**

  (**<u>not</u> True) <u>and</u> False = False** (if **<u>not</u>** is performed first)

  not (True <u>and</u> False) = True (if **<u>and</u>** is performed first)

Precedence rules to evaluate expressions

- Evaluating expressions involving scalar data items, and related operations
- **Why are "precedence rules" required?**

  FOUR examples:

  1. Consider `6 + 5 * 3` :

  – depends upon whether add '`+`' or multiply '`*`' is performed first:

  `6 + (5 * 3) = 21` (if `*` is performed first)

  `(6 + 5) * 3 = 33` (if `+` is performed first)

  2. Similarly consider `6 / 2 / 3 = 1` :

  `(6 / 2) / 3 = 1` (if the first '`/`' is performed first)

  `6 / (2 / 3) = 9` (if the second '`/`' is performed first)

  3. Or consider `6 ^ 2 / 2 :`

  `(6^2) / 2 = 18` (if '`^`' is performed first)

  `6^(2 / 2) = 6` (if '`/`' is performed first)

  4. Or consider `(not True) and False :`

  `(not True) and False = False` (if `not` is performed first)

  `not (True and False) = True` (if `and` is performed first)

- Clearly, good practice, or if this confusing:

  add parentheses to make the order of evaluation explicit

Precedence rules to evaluate expressions

- ~~BODMAS~~ Precedence rules
-
  - A precedence rule specifies which operations have higher precedence.
  - When two operations have equal precedence then whether evaluation is left to right or otherwise
  - Of course the above are overruled by use of parentheses ‘**(**‘ & ‘**)**’

- Good practice: add parentheses to make the order of evaluation explicit

Precedence rules to evaluate expressions

- **Precedence rules in Python:**
  - Parentheses have the highest precedence
    - `2 * (3-1)` is `4`
    - `(1 + 1)**(5 - 2)` is `8`
  - Exponentiation has the next highest precedence
    - `2**1 + 1` is `3` (and not `4`)
    - `3 * 1**3` is `3` (and not `27`)
  - Multiplication & division have same precedence, and higher than addition and subtraction ('+' and '-' have same precedence)
    - `2 * 3 - 1` is `5` (and not `4`)
    - `6 + 4 / 2` is `8` (and not `5`)
  - Operators with same precedence are evaluated left to right (except exponentiation, evaluated right to left)
    - `X / 2 * π` is `(X / 2) * π –` and not **X / (2 * π)**

Precedence rules to evaluate expressions

- **Precedence rules in Python:**
  - Parentheses have the highest precedence
    - `2 * (3-1)` is `4`
    - `(1 + 1)**(5 - 2)` is `8`
  - Exponentiation has the next highest precedence
    - `2**1 + 1` is `3` (and not `4`)
    - `3 * 1**3` is `3` (and not `27`)
  - Multiplication & division have same precedence, and higher than addition and subtraction ('+' and '-' have same precedence)
    - `2 * 3 - 1` is `5` (and not `4`)
    - `6 + 4 / 2` is `8` (and not `5`)
  - Operators with same precedence are evaluated left to right (except exponentiation, evaluated right to left)
    - **`X / 2 * π`** is **`(X / 2) * π`** – and not **X / (2 * π)**

Precedence rules to evaluate expressions

- Computing is about "data", and its manipulation
- Evaluating expressions involving scalar data items, and related operations
- "Precedence rules" in Python (check these rules out using interactive mode in Python):

Advice: don't try to remember these rules. If you can't tell by looking at the expression, use parentheses to make it obvious

Decreasing precedence

Decreasing precedence

```
parentheses(…)
    x**y
      -x
            x%y   x/y   x*y
              x-y   x+y
                      x==y   x!=y   x>=y   x>y   x<=y   x<y
                         not x
                          x and y x or y
```

In-class Exercise 1.2

- For Section A, follow: https://tinyurl.com/2p8ayp6y
- For Section B, follow: https://tinyurl.com/3bhthsxt

Structured data types: strings , tuples

- Composite data structures: string, tuple, list, dictionary, array, etc.
  - We will introduce these as we go along. For the present here are some examples:
  - **String**, an ordered **sequence of characters** (letters, special characters, etc.), viz.

    `{…, A, B, …, Z, a, b, …, z, 0, 1, …, 9, '.', ',','@', …}`

    Example:

    `'CSE 101 Introduction to Programming'`

    `'Mango'`

    Example operations:

    `fruit = 'Mango'`

    `fruit[0]` is `'M'`, `fruit[1]` is `'a'`

    `len(fruit)` is `5`

    `len('fruit')` is `5`

    and more, such as `'+'`, `'*'`

Structured data types: strings , tuples

- Composite data structures: string, tuple, list, dictionary, array
  - We will introduce these as we go along. For the present here are some examples:
  - **Tuple**, an ordered sequence of scalar or structured data items (including strings)

    Example of a tuple:

    ```
    ('Mahatma Gandhi', '1869/10/2')
    ```

    Example operations:

    ```
    emailTableEntry1 = ('Bijendra Jain', 'bnjain@gmail.com')
    Bapu = ('Mahatma Gandhi', (1869, 10, 2) )
    print(Bapu[1][:])
    ```

Output:

```
1869, 10, 2
```

Structured data types: lists, dictionaries, arrays

- Composite data structures: string, tuple, list, dictionary, array
  - We will introduce these as we go along. For the present here are some examples:
  - **List**, an ordered **sequence of scalar or structured data items**

    Example:

    `primes = [ 2, 3, 5, 7, 11, 13, 17, 18]`

    Example Operations :

    `len(primes)` is **`8`**

    `primes[7]` is **`18`**

    `primes[7] = 19`

    and many more
  - Dictionary, an ~~ordered~~ sequence of (key : value) pairs

    Example:

    ```
    English2Spanish = {'one': 'uno', 'three': 'tres', 'two':
    'dos'}
    ```

    Example operations:

    ```
    >>> print(English2Spanish['one'])
    >>> uno
    ```
  - Array, like vectors, matrices

    Operations: search, sort, dot-product, matrix operations, etc.

Structured data types: lists, dictionaries, arrays

- Composite data structures: string, tuple, list, dictionary, array
  - We will introduce these as we go along. For the present here are some examples:
  - **Dictionary, an ~~ordered~~ sequence of (key : value) pairs**

    Example:

    ```
    English2Spanish = {'one': 'uno', 'three': 'tres', 'two': 'dos'}
    ```

    Example operations:

    ```
    print(English2Spanish['one'])
    >>> uno
    ```

Structured data types: lists, dictionaries, arrays

- Composite data structures: string, tuple, list, dictionary, array
  - We will introduce these as we go along. For the present here are some examples:
  - **<u>Array, like vectors, matrices</u>**

    Operations: search, sort, dot-product, matrix operations, etc.

Representation of numbers

- Representation of natural nos., integers, floating point nos., Boolean
- **<u>Natural nos.: 0, 1, 2, …</u>**
  - Can only work with natural nos. limited to {0, 1, …, 2^32 **<u>-1</u>**} or {0, 1, …, 2^64 **<u>– 1</u>**} , depending upon whether we have 32 bit or 64 bit memory
    - Question: what is a ‘bit’?
  - For exam~~ple with 8 bit representation we can work only with~~ 0, 1, …, 2^8 – 1 (or 255)

0 = 0 0 0 0 0 0 0 0 , since 0 = $0*2^7 +0*2^6 +0*2^5 +0*2^4 +0*2^3 +0*2^2 +0*2^1 +0*2^0$

1 = 0 0 0 0 0 0 0 1 , since 1 = $0*2^7 +0*2^6 +0*2^5 +0*2^4 +0*2^3 +0*2^2 +0*2^1 +1*2^0$

153 = 1 0 0 1 1 0 0 1, since 153 = $1*2^7 +0*2^6 +0*2^5 +1*2^4 +1*2^3 +0*2^2 +0*2^1 +1*2^0$

Most significant bit

Least significant bit

Write an algorithm to compute the 8 bit representation of any natural no.
n ε {0, 1, 2, …, 255)

https://profile.iiita.ac.in/bibhas.ghoshal/lecture_slides_coa/Data_Representation.html

Representation of integers

- Representation of (signed) **Integers**: …, -2, -1, 0, 1, 2, …
  - With 32 bit representation, signed integer is best written in "2's complement" notation
  - Can represent integers in {-2^31, …, -2, -1, 0, 1, 2, … , 2^31 **-1**}
  - For example with 8 bit representation, MSB bit 7 is sign bit **0** for '+', **1** for '–'
  - Bits 6 through 0 essentially give the magnitude

    {-2^7, …, 2, 1, 0, 1, 2, … , 2^7 -1} or {-12**8**, …, -2, -1, 0, 1, 2, … 12**7**}

    To be sure:

    +127 = **0** 1 1 1 1 1 1 1

    …

    +2 = **0** 0 0 0 0 0 1 0

    Add 1

    +1 = **0** 0 0 0 0 0 0 1

    Add 1

    +0 = **0** 0 0 0 0 0 0 0

    Add 1

    -1 = **1** 1 1 1 1 1 1 1

    Add 1

    -2 = **1** 1 1 1 1 1 1 0

    …

-**128** = **1** 0 0 0 0 0 0 0 (pl. check)

https://en.wikipedia.org/wiki/Two%27s_complement

Representation of floating point numbers

- Representation of natural nos., integers, floating point nos., Boolean
- **Floating point** nos., such -4.5
  - Will have two constraints:
    - Range: Or how large can the no. be?
    - Accuracy: Or how accurately can the no. be represented?:
  - 32-bit single precision vs. 64-bit double precision (how is range, accuracy impacted?)
  - Even a 64-bit double precision representation is an approximation
    - Consider representing 1/3 or π

May also read: https://www.geeksforgeeks.org/floating-point-representation-basics/

Representation of floating point numbers

- Representation of natural nos., integers, **floating point nos**., Boolean
  - Question: how is -4.5 represented:
    - 'decimal' notation : $-4.5_{(10)}$
    - 'base 2' notation: $-100.1_{(2)}$

$= -(4*10^{0} +5*10^{-1})$

$= -(1*2^{2} +0*2^{1} +0*2^{0} +1*2^{-1})$

May also read: https://www.geeksforgeeks.org/floating-point-representation-basics/

Representation of floating point numbers

- Representation of natural nos., integers, **floating point nos**., Boolean
  - Question: how is -4.5 represented:
    - 'decimal' notation : $-4.5_{(10)}$
    - 'base 2' notation: $-100.1_{(2)} = -1.001_{(2)} \times 2^{+2}$

$= -(4*10^{0} +5*10^{-1})$

$= -(1*2^{2} +0*2^{1} +0*2^{0} +1*2^{-1})$

May also read: https://www.geeksforgeeks.org/floating-point-representation-basics/

Representation of floating point numbers

- Representation of natural nos., integers, **floating point nos**., Boolean
  - Question: how is -4.5 represented:
    - 'decimal' notation : $-4.5_{(10)}$
    - 'base 2' notation: $-100.1_{(2)} = -1.001_{(2)} \times 2^{+2}$

$= -(4*10^{0} + 5*10^{-1})$

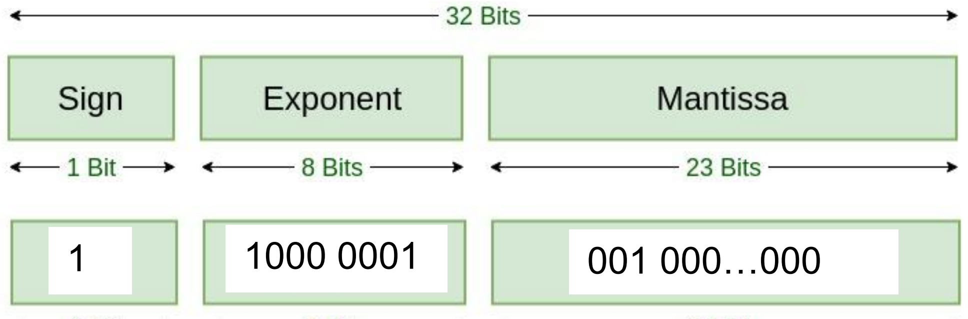
$= -(1*2^{0} + 0*2^{-1} + 0*2^{-2} + 1*2^{-3}) * 2^{2}$

$= -(1*2^{2} + 0*2^{1} + 0*2^{0} + 1*2^{-1})$

May also read: https://www.geeksforgeeks.org/floating-point-representation-basics/

Representation of floating point numbers

- Representation of natural nos., integers, floating point nos., Boolean
  - Question: how is -4.5 represented:
    - ‘decimal’ notation : $-4.5_{(10)}$
    - ‘base 2’ notation: $-100.1_{(2)} = -1.001_{(2)} \times 2^{+2}$



  - The exponent is in ‘excess 127’ notation, 127 + exponent
  - +1.0 value is represented as:

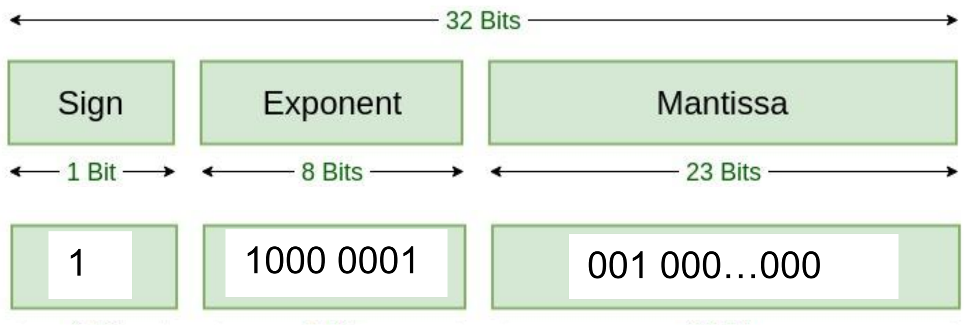
    +0 = 0 01111111 00000000000000000000000
  - A zero value and has two special representations:

    +0 = 0 00000000 00000000000000000000000, or -0 = 1 00000000 00000000000000000000000

May also read: https://www.geeksforgeeks.org/floating-point-representation-basics/

Representation of floating point numbers

- Representation of natural nos., integers, floating point nos., Boolean
  - Question: how is -4.5 represented:
    - 'decimal' notation : $-4.5_{(10)}$
    - 'base 2' notation: $-100.1_{(2)} = -1.001_{(2)} \times 2^{+2}$



  - The exponent is in 'excess 127' notation, 127 + exponent
  - +1.0 value is represented as:

    +0 = 0 01111111 00000000000000000000000
  - A zero value and has two special representations:

    +0 = 0 00000000 00000000000000000000000, or -0 = 1 00000000 00000000000000000000000

May also read: https://www.geeksforgeeks.org/floating-point-representation-basics/

In-class Exercise 1.3

- For Section A, follow: https://tinyurl.com/ms2p2t4v

- For Section B, follow: https://tinyurl.com/4ewdb95h

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