Week-1: Basic programming concepts

Sallar Khan

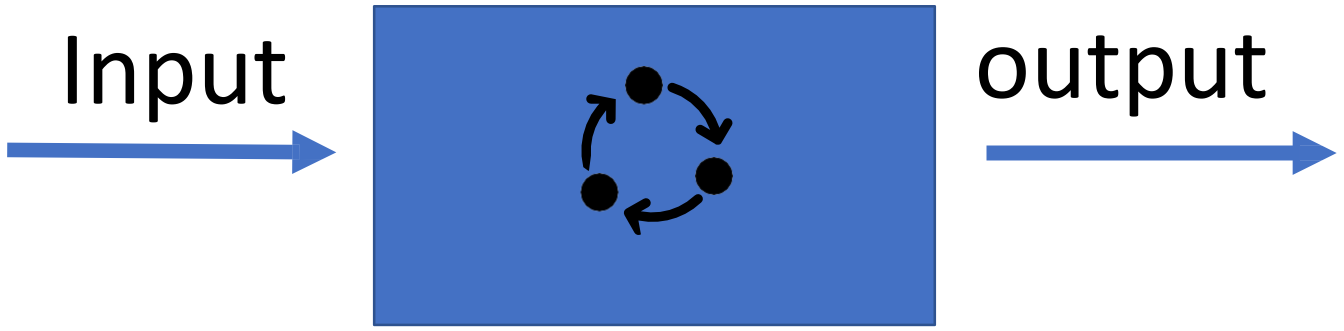
Sallar.khan@ncirl.ie

Outline

- Computer Program
- Data Types
  - Number, Bool,
  - Sequence Types,
  - Mapping, and
  - Sets
- Type Casting
- Operators & Operations
  - Arithmetic, Comparison, and Sequence operators & Sequence operations
  - Operators for Sets & Dictionaries
  - Practice & Exercise
- Control Flow
- Conditions & Loops
- Practice & Exercise

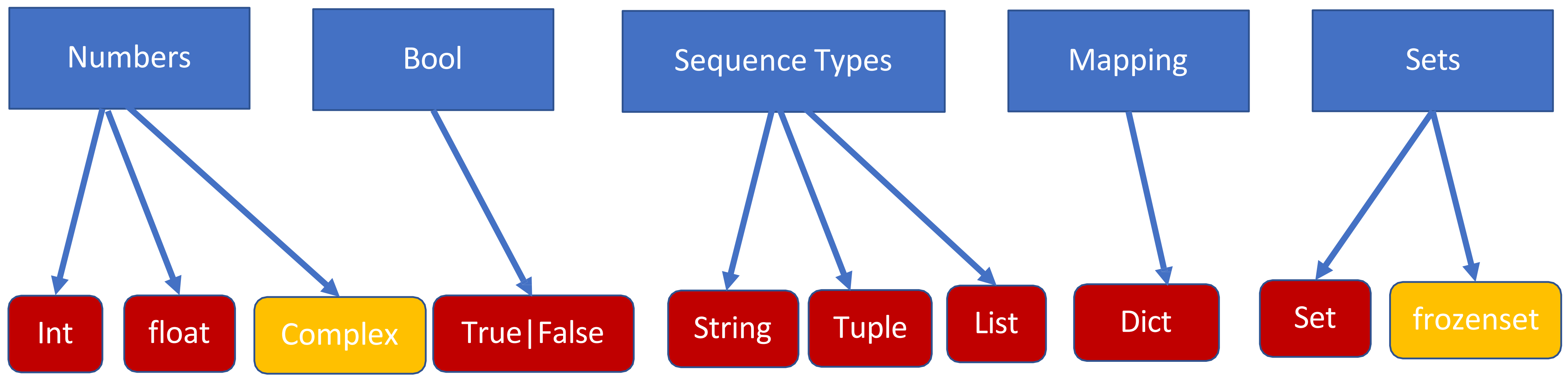
Computer Program

- A **program**, also called an **application** or **software**, is a set of instructions that process input, manipulate data, and output a result [1].



[1] https://www.computerhope.com/jargon/p/program.htm

Data Types



Some Others: None Type & Bytes etc.

Python related information

- Identifier names (rules)
  - Reserved words
    - Class, int, for, while etc.,
- Loosely typed language
  - No need to declare identifiers before usage
- Case Sensitive
  - x =1 & X = 3 (two different identifiers)

Numbers & Bool

- Bool
  - myVar = True | False
  - `print(myVar)` => True|False
- Integer
  - x = 0
  - `print(x)` => 0
- Float
  - x = 0.9 => 0.9
  - `print(x)` => 0.9
- Applications: you will need these almost in every situation

Sequence Types (String)

- message = “hello” or message = ‘hello’
- `print(message)` => hello
- String is a sequence
  - `print(message[2])` => l
- Multi-line string

  -
    ```
    print(“““Welcome to the GPA calculator.
      Please enter all your letter grades, one per line.
      Enter a blank line to designate the end.”””)
    ```
    or ‘‘‘string ...... ’’’
- Escape Character required:
  - ‘Don\’t worry’
  - path = “C:\\Python\\” and then `print(path)` => C:\Python\
- Immutable
  - greeting = 'Hello, world!'
  - greeting[0] = ‘J’ => TypeError: 'str' object does not support item assignment

Sequence Types (List)

- A list is a sequence of values. In a string, the values are characters; in a list, they can be **any type**.
- Square brackets “[" and “]” are used to create list
- Examples:
  - `list1 = [10, 20, 30, 40]`
  - `print(list1) => [10, 20, 30, 40]`
  - `list2 = ['nci', 'tcd', 'dcu']`
  - `print(list2) => ['nci', 'tcd', 'dcu']`

Sequence Types (List) - Continued

- Multiple data types & nested lists
- `list1 = ['spam', 2.0, 5, [10, 20]]`

```
print(list1[0]) => spam
print(list1[3]) => [10,  20]
print(list1[3][0]) => 10
```

- IndexError: list index out of range, when e.g., print(list1[10])
- Negative index => values from the end, e.g., `print(list1[-1]`
- **Mutable**

```
     list1 [1] = 23 (OK)
```

Sequence Types (Tuple)

- A tuple is a sequence of values much like a list.
- The values stored in a tuple can be any type, and they are indexed by integers.
- The important difference is that tuples are **immutable**.
- t = 'a', 'b’, 2, 'd', 'e’ **or** with enclosed with in parenthesis **()**
- **Tuple with single element: (‘a’,)**
- Accessing elements: print(t[0]) => a
- t[2] = 2 => **TypeError**: 'tuple' object does not support item assignment

Dictionary

- Associative Array (key-value=> pair)
- In comparison to list the **index positions** have to be integers; in a dictionary, the indices can be (almost) any type.
-
```
eng2sp = {'one': 'uno’,
          'two': 'dos’,
          'three': 'tres’
          }
```
- `print(eng2sp['two’])` => dos
- **mutable**

Sets

- A set is a collection which is **unordered**, **unchangeable***, and **unindexed**.
- `myset = {20, "Ireland", True}`
- **Unindexed =>** `print(myset[0]) TypeError: 'set' object is not subscriptable`
- **Unchangeable =>** `myset[0]=“new value” TypeError: 'set' object does not support item assignment`
- Sets cannot have duplicate items.
- You can add or remove items, will see that later

Type Casting

- Type casting or type conversion: It is the process of converting one data type to another data type.
- type function get type, e.g.,
  - `x = 10`
  - `type (x) => <class 'int'>`
- int(), float(), str(), tuple(), set(), list(), dict()
- For example:
- `int(‘32’) => 32`
- `int('Hello’) => ValueError: invalid literal for int() with base 10: 'Hello’`
- `int(3.99999) => 3`
- `int(-2.3) => -2`

Operators

- Logical Operators (**not, and, or)**
- Equality Operators (==, !=, **is** same identity, **is not** different identity)
- Comparison Operators (<,<=, >, and >=)
- Arithmetic Operators(+, −, *, / true division, // integer division, % the modulo operator)
- true division => `27 / 4` => 6.75
- integer division => `27 // 4` => 6

Sequence Operators

- The sequence types (str, tuple, and list) support the following operator syntaxes
  - `s[j]` element at index j
  - `s[start:stop]` slice including indices [start,stop)
  - `s[start:stop:step]` slice including indices start, start + step,
    - `start + 2 step,...,` up to but not equalling or stop
  - `s + t` concatenation of sequences
  - `K*s` shorthand for `s + s + s + ... (k times)`
  - `val` **`in`** `s` containment check
  - `val` **`not in`** `s` non-containment check

Sequence Operations

- `s == t` equivalent (element by element)
- `s != t` not equivalent
- `s < t` lexicographically less than
- `s <= t` lexicographically less than or equal to
- `s > t` lexicographically greater than
- `s >= t` lexicographically greater than or equal to

Operators for Sets and Dictionaries

- `key in s` containment check
- `key not in s` non-containment check
- `s1 == s2` `s1` is equivalent to `s2`
- `s1 != s2` `s1` is not equivalent to `s2`
- `s1 <= s2` `s1` is subset of `s2`
- `s1 < s2` `s1` is proper subset of `s2`
- `s1 >= s2` `s1` is superset of s2
- `s1 > s2` `s1` is proper superset of `s2`
- `s1 | s2` the union of `s1` and `s2`
- `s1 & s2` the intersection of `s1` and `s2`
- `s1 – s2` the set of elements in `s1` but not `s2`

Let's Practice

Control flow – If-else condition

```
if first condition:
    first body
elif second condition:
    second body
else:
    third body
```

While Loops

```
while condition:
    body
```

Example:

```
j = 0
while j < len(data) and data[j] != X :
    j += 1
```

For Loops

```
for element in iterable:
    body
```

Example: Task of computing the sum of a list of numbers.

```
total = 0
for val in data:
    total += val
```

Example: Maximum value in a list of elements

```
total = 0
for val in data:
    total += val
```

Index-Based For Loops

- `range(n)` generates the series of *n* values from 0 to *n* - 1.

```
big_index = 0
for j in range(len(data)):
if data[j] > data[big_index]:
    big_index = j
```

Break and Continue Statements

- **break** statement immediately terminate a while or for loop when executed within its body.
- **continue** statement that causes the current iteration of a loop body to stop, but with subsequent passes of the loop proceeding as expected.

```
found = False
for item in data:
if item == target:
       found = True
       break
```

```
oddSum = 0
for item in data:
 if item % 2 != 0:
 continue
 oddSum =  oddSum + item
```