COMP10001

WORKSHOP #3

Shevon Mendis
shevonm@unimelb.edu.au

Booleans

- A Boolean, or `bool` in python, is a data type that stores a truth value, ie. `True` or `False`.

Booleans

- A Boolean, or `bool` in python, is a data type that stores a truth value, ie. `True` or `False`.
- Other types can be converted into it by using the `bool()` function.

Booleans

- A Boolean, or `bool` in python, is a data type that stores a truth value, ie. `True` or `False`.
- Other types can be converted into it by using the `bool()` function.
    - With `int`s: `0` converts to `False` while all other values convert to `True`

Booleans

- A Boolean, or `bool` in python, is a data type that stores a truth value, ie. `True` or `False`.
- Other types can be converted into it by using the `bool()` function.
    - With `int`s: `0` converts to `False` while all other values convert to `True`
    - With `float`s: `0.0` converts to `False` while all other values convert to `True`

Booleans

- A Boolean, or `bool` in python, is a data type that stores a truth value, ie. `True` or `False`.
- Other types can be converted into it by using the `bool()` function.
    - With `int`s: `0` converts to `False` while all other values convert to `True`
    - With `float`s: `0.0` converts to `False` while all other values convert to `True`
    - With `str`s: The empty string, `""`, converts to `False` while all other (non-empty) string sequence convert to `True`

Relational Operators

- Compares two values and produces a boolean result

Operator	Meaning
`==`	Equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`!=`	Not equal to

Logical Operators

- Enforces logical constraints on its operands. They combine multiple boolean values into a single truth value.

Logical Operators

- Enforces logical constraints on its operands. They combine multiple boolean values into a single truth value.

- Includes:

    - `And`
        - Requires **all** operands to be `True` for the entire statement to be `True`; `False` otherwise.

Logical Operators

- Enforces logical constraints on its operands. They combine multiple boolean values into a single truth value.

- Includes:
    - `And`
        - Requires **all** operands to be `True` for the entire statement to be `True`; `False` otherwise.
        - `True and True` → `True`
        - `True and False` → `False`
        - `False and True` → `False`
        - `False and False` → `False`

Logical Operators

- `or`
  - Requires **at least one** of the operands to be `True` for the entire statement to be `True`; `False` otherwise.

Logical Operators

- `or`
  - Requires **at least one** of the operands to be `True` for the entire statement to be `True`; `False` otherwise.
  - `True or True` → `True`
  - `True or False` → `True`
  - `False or True` → `True`
  - `False or False` → `False`

Logical Operators

- `or`
  - Requires **at least one** of the operands to be `True` for the entire statement to be `True`; `False` otherwise.
  - `True or True` → `True`
  - `True or False` → `True`
  - `False or True` → `True`
  - `False or False` → `False`
- `not`
  - Inverts a truth value.

Logical Operators

- `or`
  - Requires **at least one** of the operands to be `True` for the entire statement to be `True`; `False` otherwise.
  - `True or True` → `True`
  - `True or False` → `True`
  - `False or True` → `True`
  - `False or False` → `False`
- `not`
  - Inverts a truth value.
  - `not True` → `False`
  - `not False` → `True`

Order of Precedence

- The *Order of Precedence* is Relational Operators, then **`not`**, then **`and`**, finally, **`or`**.

Order of Precedence

- The *Order of Precedence* is Relational Operators, then `not`, then `and`, finally, `or`

- When the precedence of relational or logical operators is equal, *chaining* occurs, where the sub-conditions are evaluated simultaneously.
  - eg. `50 < num <= 100` is evaluated as `(50 < num) and (num <= 100)`

Order of Precedence

- The *Order of Precedence* is Relational Operators, then `not`, then `and`, finally, `or`.

- When the precedence of relational or logical operators is equal, *chaining* occurs, where the sub-conditions are evaluated simultaneously.
  - eg. `50 < num <= 100` is evaluated as `(50 < num) and (num <= 100)`

- Brackets can be used to clarify the order of operations as well.

if Statements

- Skeleton of an if statement:

```
if < condition > :
    # do something
elif < condition > :
    # do something else
else:
    # do something else
```

Sequences

- A sequence is a data type which allows us to store a series of objects **in a particular order.**
- `str`s store sequences of characters while `list`s and `tuple`s can store sequences of any type of object.

Indexing

- Indexing means accessing the element stored in a particular (integer) position in a **sequence.**

Indexing

- Indexing means accessing the element stored in a particular (integer) position in a **sequence.**

- **eg.** `my_list = [1, 1, 2, 3, 5, 8]`

```
                ↑  ↑  ↑  ↑  ↑  ↑
                0  1  2  3  4  5
               -6 -5 -4 -3 -2 -1
```

Indexing

- Indexing means accessing the element stored in a particular (integer) position in a **sequence.**

- **eg.** `my_list = [1, 1, 2, 3, 5, 8]`

```
                    ↑  ↑  ↑  ↑  ↑  ↑
                    0  1  2  3  4  5
                   -6 -5 -4 -3 -2 -1
```

- Trying to access an element at an index that doesn't exist results in an **`IndexError`**.

Slicing

- Allows us to *slice* a subsection of a **sequence.**

Slicing

- Allows us to *slice* a subsection of a **sequence.**

- Format:
  - <variable_name or object_literal> **`[start_index : stop_index : step_size]`**

Slicing

- Allows us to *slice* a subsection of a **sequence.**

- Format:
    - <variable_name or object_literal> `[start_index : stop_index : step_size]`
    - *start_index* : index to start slicing at (included in the slice)
    - *stop_index* : index to stop slicing at (excluded in the slice)
    - *step_size* : number of elements to move over by when slicing

Slicing

- Things to note:
  - Slicing **always** returns a sequence (**never** produces an `IndexError`)

Slicing

- Things to note:
    - Slicing **always** returns a sequence (**never** produces an `IndexError`)
    - If the *start_index* is not explicitly defined, it defaults to 0*
    - If the *stop_index* is not explicitly defined, it defaults to the length of the sequence*
    - If the *step_size* is not explicitly defined, it defaults to 0

*assuming that the slicing direction is left to right

Functions

- Skeleton:

```
def function_name(arg1, arg2):
    # do something
    return something
```

Functions

- Skeleton:

```python
def function_name(arg1, arg2):
    # do something
    return something
```

- Using `return` is optional. By default, a function will return `None`.

Functions

- Skeleton:

```python
def function_name(arg1, arg2):
    # do something
    return something
```

- Using `return` is optional. By default, a function will return `None`.
- Brackets are needed to *call* a function. Without brackets, we're just using the function's name as a *reference*.

Functions

- Advantages:
    - Reduces code duplication/ Increases modularity
    - Increases code maintainability