15-110 Principles of Computing – F19

Lecture 14:
Iteration 1

Teacher:
Gianni A. Di Caro

So far about Python ...

- Basic elements of a program:
  - Literal objects
  - Variables objects
  - Function objects
  - Commands
  - Expressions
  - Operators

- Utility functions (built-in):
  - `print(arg1, arg2, …)`
  - `type(obj)`
  - `id(obj)`
  - `int(obj)`
  - `float(obj)`
  - `bool(obj)`
  - `str(obj)`
  - `input(msg)`
  - `len(non_scalar_obj)`
  - `sorted(seq)`
  - `min(seq), max(seq)`

- Object properties
  - Literal vs. Variable
  - Type
  - Scalar vs. Non-scalar
  - Immutable vs. Mutable
  - Aliasing vs. Cloning

- Conditional flow control

```
if cond_true:
    do_something
```

```
if cond_true:
    do_something
else:
    do_something_else
```

```
if cond1_true:
    do_something_1
elif cond2_true:
    do_something_2
else:
    do_something_else
```

- Data types:
  - `int`
  - `float`
  - `bool`
  - `str`
  - `None`
  - `tuple`
  - `list`

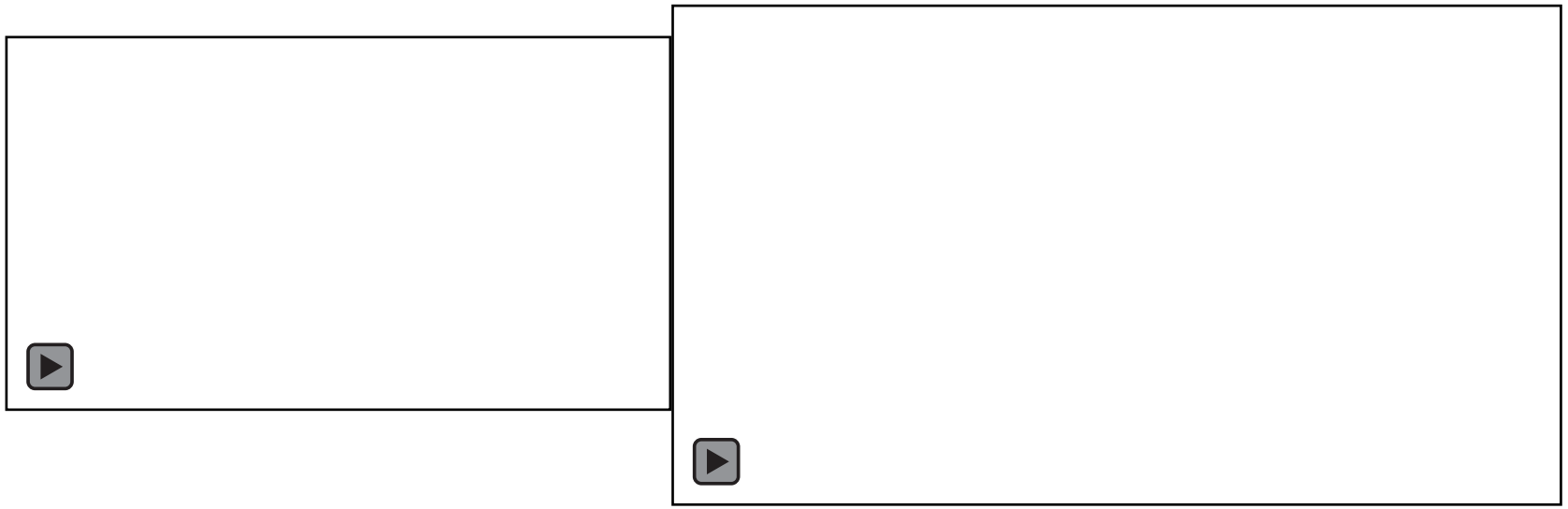
- Relational operators
  - `>`
  - `<`
  - `>=`
  - `<=`
  - `==`
  - `!=`

- Logical operators
  - `and`
  - `or`
  - `not`

- Operators:
  - `=`
  - `+`
  - `+=`
  - `-`
  - `/`
  - `*`
  - `*=`
  - `//`
  - `%`
  - `**`
  - `[]`
  - `[:]`
  - `[::]`

- String methods
- List methods

Repeating actions



Constructs and Operators for *iterations* in python:

***for* loop**

```
for variable in sequence:
    actions
```

Definite loop

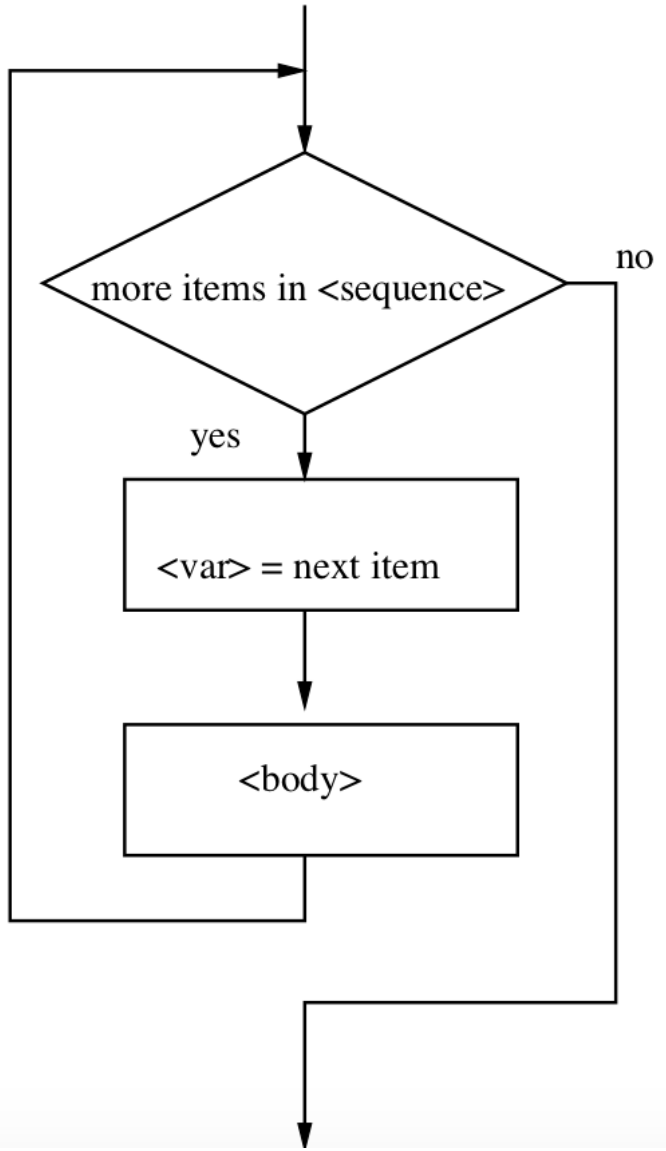
***while* loop**

```
while condition_is_true:
    actions
```

Indefinite loop

Definite loops: `for` construct

- ✓ Repeat a set of actions a **defined number of times** (at *most*)
- ✓ Each time the action *can* be executed on a different input parameter



Definite loops: `for` construct

```
for variable in sequence:
    actions
```

scope of for loop (indent)

sequence: tuple, list

`len(sequence)` **iterations** (at most)

variable: *loop index*: each time the variable is set to the value of the next item in the sequence

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

```
6 6
9 3
14 5
21 7
```

Definite loops: for construct

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

```
6 6
9 3
14 5
21 7
```

Enrolled loop

n = 6	n = 3	n = 5	n = 7
sum += n	sum += n	sum += n	sum += n
Iteration 1 (sum is 6)	**Iteration 2** (sum is 9)	**Iteration 3** (sum is 14)	**Iteration 4** (sum is 21)

Counting, selecting, summing in `for` loops, `sum()` function

- A typical use of loops is for **counting items** that satisfy certain conditions

```
data_list = [1, 3, -5, 7, -11]
counter = 0
for i in data_list:
    if i < 0:
        counter += 1
```

`counter`: accumulator

```
data_list = [1, 'red', -5, 'g', -11]
counter = 0
for i in data_list:
    if type(i) == str:
        counter += 1
```

Counting, selecting, summing in `for` loops, `sum()` function

- A typical use of loops is for **selecting items** that satisfy certain conditions

```
data_list = [1, 3, -5, 7, -11]
positive_numbers = []
for i in data_list:
    if i > 0:
        positive_numbers.append(i)
```

```
data_list = [1, 'red', -5, 'green', -11]
colors = []
index = 0
for x in data_list:
    if type(x) == str:
        colors.append( (index, x) )
    index += 1
```

Counting, selecting, summing in `for` loops, `sum()` function

- Another typical use of loops is for **summing up item values** (e.g., that satisfy certain conditions)

```
data_list = [3, -1, 0, 1, 4]
data_sum = 0
for i in data_list:
    data_sum += i
```

```
data_list = [3, -1, 0, 1, 4]
data_sum = 0
for i in data_list:
    if i > 0:
        data_sum += i
```

- `sum(l)` function does the same thing: sums up the element of the list `l`

```
data_list = [(3,2), (-1, -3), (0,-2), (1,1), (4,6)]
sum = 0
for i in data_list:
    if i[1] > 0:
        sum += i[0]
```

What about `sum()` here?

Any operation that needs to be iterated

- Any arithmetic operation of interest that can be iteratively applied (e.g., **factorial)**

```
factorial = 1
for n in [1, 2, 3, 4]:
   factorial *= n
print('Factorial of', n, 'is',  factorial)
```

`for` loops: use of loop variable

- So far, loop actions have made a **direct use of the variable value**
- The variable plays the role of a *changing input parameter*

```
factorial = 1
for n in [1, 2, 3, 4]:
    factorial *= n
print('Factorial of', n, 'is',  factorial)
next_factorial = factorial * (n + 1)
```

➢ **Loop index variable:**

✓ the variable created by the for loop doesn't disappear after the loop is done

✓ it will contain the last value used in the `for` loop

`for` loops: use of loop variable

```
colors = ['r', 'g', 'y', 'b', 'g', 'bk']
count_green = 0
for c in colors:
    if c == 'g':
        print('A green item')
        count_green += 1
        print('Found', count_green, 'green items')
print('Last color checked was', c)
```

- **Loop index variable**:
  - ✓ the variable created by the for loop doesn't disappear after the loop is done
  - ✓ it will contain the last value used in the `for` loop

`for` loops: getting a range of numbers, `range()` function

- Sometimes, a loop **doesn't need to to make a direct use of the variable value**, we might only need to specify how many iterations should be performed: we only need an **iteration counter**

```
for i in [1, 2, 3, 4, 5]:
    print('Hello! ')
```

```
sum_up = 0
for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    sum_up += 1
```

Only need to say how many times the action should be iterated (5, 10)

- More in general, it might be useful to **automatically generate lists of numbers (integer)**, that can be also used as input parameters during iterations → `range(start, end, step)` function
- **Counted loops**

`for` loops: getting a range of numbers, `range()` function

- `range(start, end, step)` generates the integer numbers in the specified range
  - `start, end, step` must be **integer**
  - range is *exclusive*: the **last number is not generated**

```
for i in range(-1,10,2):
    print('Counter:',i)
```

→ -1, 1, 3, 5, 7, 9

✓ `range(s, n, ss)` generates the integers between `s` and `n-1` with a step of `ss`

```
for i in range(2,9):
    print('Counter:',i)
```

→ 2, 3, 4, 5, 6, 7, 8

✓ `range(s, n)` is equivalent to `range(s, n, 1)`

✓ generates the successive integers between `s` and `n-1`

```
for i in range(10):
    print('Counter:', i)
```

→ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

✓ `range(n)` is equivalent to `range(0, n, 1)`

✓ generates the successive integers between `0` and `n-1`

`for` loops: getting a range of numbers, `range()` function

- `range(s, n, ss),` **rules for the arguments:**
  - **Increasing ranges**: `n > s`, `ss` must be *positive*

```
for i in range(1,5,1):
    print('Counter:',i)
```
→ 1, 2, 3, 4

```
for i in range(1,5,-1):
    print('Counter:',i)
```
→ Nothing

```
for i in range(-10):
    print('Counter:',i)
```
→ Nothing

  - **Decreasing ranges**: `s > n`, `ss` must be *negative*

```
for i in range(5,1,-1):
    print('Counter:',i)
```
→ 5, 4, 3, 2

```
for i in range(10, 1):
    print('Counter:',i)
```
→ Nothing

```
for i in range(5,1,1):
    print('Counter:',i)
```
→ Nothing

`for` loops: getting a range of numbers, `range()` function

- Watch out: `range()` **doesn't generates all numbers at once**
- **i.e., it doesn't return a list with the numbers!**
- Range is a **generator**

```
numbers = range(5)
print(len(numbers))          →  5
print(numbers[2])            →  2
print((2 in numbers))        →  True
for i in numbers:            →  0, 1, 2, 3, 4
   print('Counter:', i)
```

it *looks* like a list, the behavior is the expected one …

`for` loops: getting a range of numbers, `range()` function

```
numbers = range(5)
print(numbers)
```

→ range(0, 5)

- **Generator:** It's a sequence because the object supports membership testing, indexing, slicing and has a length, just like a list or a tuple.
- Unlike a list or a tuple, it doesn't actually contain all integers in the sequence in memory, making it *virtual*, elements are returned on demand

```
numbers = list(range(5))
print(numbers)
```

→ 0, 1, 2, 3, 4