# Computer Programming I

Binnur Kurt, PhD

BAU Bahçeşehir University

binnur.kurt@rc.bau.edu.tr

MODULE 6

## CONTROL FLOW

## Control Flow

> Python has several built-in keywords for conditional logic, loops, and other standard *control flow* concepts found in other programming languages.
  - if, elif, and else
  - for loops
  - while loops
  - pass
  - range
  - ternary expressions

## `if`, `elif`, and `else`

> The `if` statement is one of the most well-known control flow statement types.

> It checks a condition that, if `True`, evaluates the code in the block that follows:

```python
if x < 0:
    print('It's negative')
```

## if, elif, and else

> An if statement can be optionally followed by one or more elif blocks and a catch-all else block if all of the conditions are False:

```
if x < 0:
    print('It's negative')
elif x == 0:
    print('Equal to zero')
elif 0 < x < 5:
    print('Positive but smaller than 5')
else:
    print('Positive and larger than or equal to 5')
```

## if, elif, and else

> If any of the conditions is True, no further elif or else blocks will be reached.

> With a compound condition using and or or, conditions are evaluated left to right and will short-circuit:

```
In [117]: a = 5; b = 7

In [118]: c = 8; d = 4

In [119]: if a < b or c > d:
   .....:     print('Made it')
Made it
```

## `for` loops

> `for` loops are for iterating over a collection (like a list or tuple) or an iterator.

> The standard syntax for a `for` loop is:

```
for value in collection:
    # do something with value
```

## `for` loops

> You can advance a `for` loop to the next iteration, skipping the remainder of the block, using the continue keyword.

> Sum up integers in a list and skips `None` values:

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
    if value is None:
        continue
    total += value
```

## `for` loops

> A `for` loop can be exited altogether with the `break` keyword.

> Sum up elements of the list until a 5 is reached:

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
    if value == 5:
        break
    total_until_5 += value
```

## `for` loops

> `break` keyword only terminates the innermost `for` loop

> Any outer for loops will continue to run:

```
In [121]: for i in range(4):
   .....:     for j in range(4):
   .....:         if j > i:
   .....:             break
   .....:         print((i, j))
   .....:
(0, 0)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
(3, 3)
```

## while loops

> A while loop specifies a condition and a block of code that is to be executed

– until the condition evaluates to False

or

– the loop is explicitly ended with break

```
x = 256
total = 0
while x > 0:
    if total > 500:
        break
    total += x
    x = x // 2
```

## break and continue Statements

> The break and continue statements alter a loop's flow of control.

> Executing a break statement in a while or for immediately exits that statement.

```
for number in range(100):
    if number == 10:
        break
    print(number, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

```
for number in range(100):
    if number == 10:
        continue
    print(number, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
```

## `pass`

> `pass` is the “no-op” statement in Python.

> It can be used in blocks where

– no action is to be taken

– a placeholder for code not yet implemented

> It is only required because Python uses whitespace to delimit blocks:

```python
if x < 0:
    print('negative!')
elif x == 0:
    # TODO: put something smart here
    pass
else:
    print('positive!')
```

## `range`

> The `range` function returns an iterator that yields a sequence of evenly spaced integers:

```
In [122]: range(10)
Out[122]: range(0, 10)

In [123]: list(range(10))
Out[123]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

> Both a start, end, and step (which may be negative) can be given:

```
In [124]: list(range(0, 20, 2))
Out[124]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [125]: list(range(5, 0, -1))
Out[125]: [5, 4, 3, 2, 1]
```

## range

> **range** produces integers up to but not including the endpoint.

> A common use of **range** is for iterating through sequences by index:

```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
    val = seq[i]
```

## range

> While you can use functions like **list** to store all the integers generated by **range** in some other data structure, often the default iterator form will be what you want.

> This snippet sums all numbers from 0 to 99,999 that are multiples of 3 or 5:

```
sum = 0
for i in range(100000):
    # % is the modulo operator
    if i % 3 == 0 or i % 5 == 0:
        sum += i
```

> While the range generated can be arbitrarily large, the memory use at any given time may be very small.

## Ternary expressions

> A *ternary expression* in Python allows you to combine an if-else block that produces a value into a single line or expression.

> The syntax for this in Python is:

value = *true-expr* ***if*** *condition* ***else*** *false-expr*

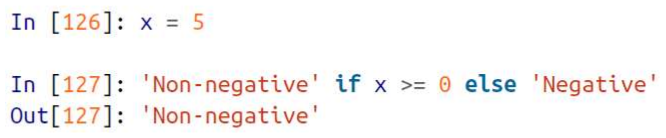
– *true-expr* and *false-expr* can be any Python expressions.

> It has the identical effect as the more verbose:

```
if condition:
  value = true-expr
else:
  value = false-expr
```

## Ternary expressions

> Example



```
In [126]: x = 5

In [127]: 'Non-negative' if x >= 0 else 'Negative'
Out[127]: 'Non-negative'
```