5: Repetitive execution

The basic blocks in programs

- Input:
  - Get data from user
- Output:
  - Send messages to user
- Conditional execution:
  - Execute different blocks based on conditions

The problem

- If there are lot of steps to handle, writing each statement one at a time is not practical.

- Solution

  - Repetitive execution: execute the program block as long as needed

Repetition is needed to...

- Handle sequences of unknown or variating length
- Repeat a step as long as the user wants to
- Save time and effort when handling longer sequences
- à very important in most programs

Computers are good at loops

- In fact, repetition is one of the main reasons to write programs

- Computers are not very good at complex operations – they are however very good at repeating simple operations very quickly over and over again.

- …which is often the best way to perform complex operations… J

Repetiton in programming

- Typically:
  - **Repeat a program block** while condition is True (or until the condition is True)

  - Instead of performing the same exact operation over and over, **variable or variables are usually changed inside the loop**

  - This will finally lead to **terminating condition**, which ends the loop execution

Repetition in Python

- Two constructs: **while** and **for**

- Main difference: **for** statement is to be used when iterating through a *sequence*, **while** is more useful when the condition is more complex or the termination point can not be pre-defined (e.g. it depends on user input)

While loop

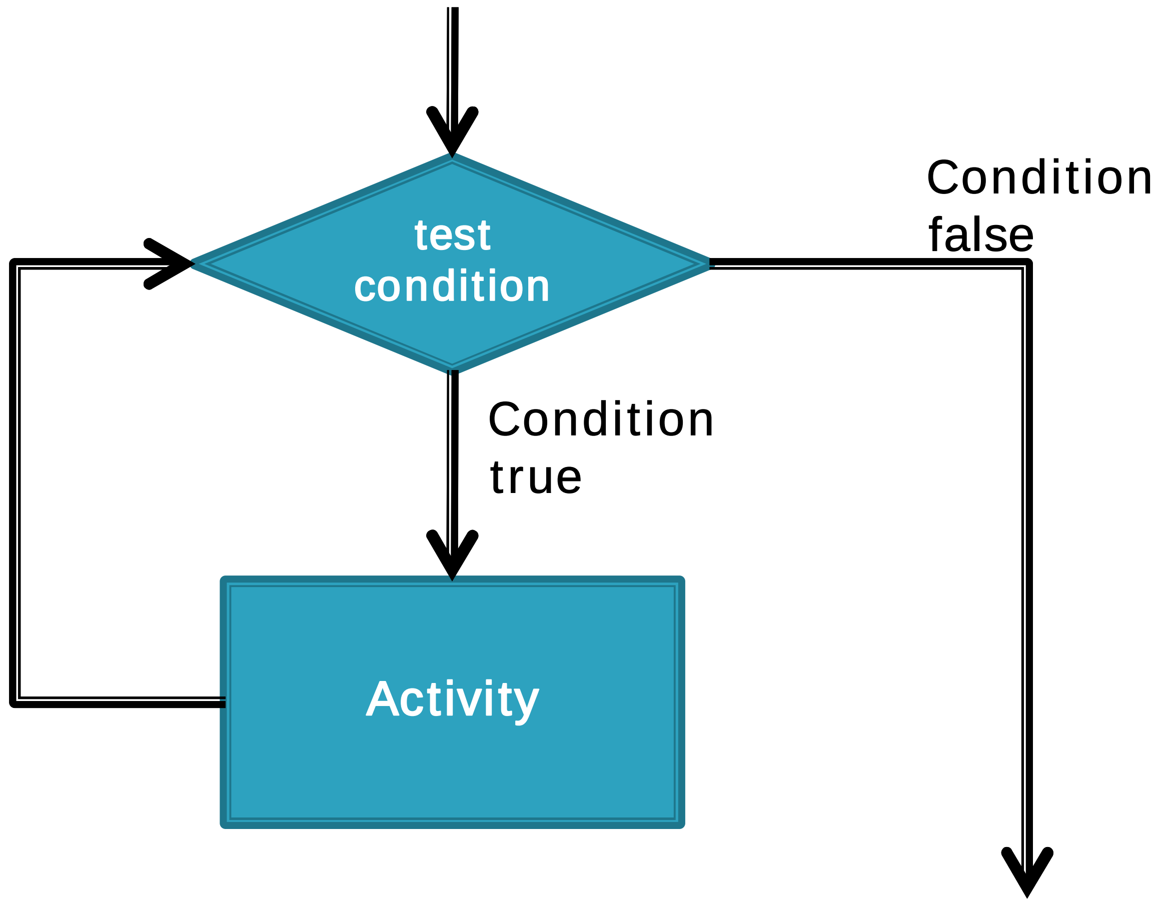
- The syntax of the while loop in Python:

```
while condition:
   statement block
```

- The statement block is executed *as long as the condition evalutes to True*.

- Note, that the statement block may not be executed at all, if the condition is false to begin with (a pre-test loop).

While loop in Python (cont.)



While loop in Python (cont.)

- Hence, the loop starts by checking the condition
- If condition evaluates into True, the **loop body** (i.e. the statement block) is executed
- After this, the condition is checked again
- This goes on until the condition becomes False or the loop is forced to terminate

Example

- Output numbers 1...10:

```
number = 1
while number <= 10:
    print number
    number = number + 1
```

Checking the condition

- A common mistake is to assume that the loop is terminated immediately after the state used in condition becomes false

- However, the condition is only checked **after** the loop body (i.e. the statement block is executed)

For example

```
number = 1
printMore = True
while printMore:
  print number
  number = number + 1
  if number == 3:
     printMore = False
  print "Still printing..."
```

Three components of a loop:

- **Initialize –** Establish an initial state that will be modified toward the termination condition
- **Test –** Compare the current state to the termination condition and terminate the repetition if equal
- **Modify –** Change the state in such a way that it moves toward the terminate condition

Example (again):

- Output numbers 1...10:

```
number = 1
while number <= 10:
  print number
  number = number + 1
```

Initialize

Test

Modify

Example 2

- Calculate the powers of two until user enters a negative number:

```
number = 0
while number >= 0:
    number = input("Give a number: ")
    print "Number ^ 2 =", number ** 2
```

Infinite loops

- If one of the three components of a loop is missing (or not valid), an **infinite loop** may occur:

```
number = 1
while number <= 10:
    print number
```

- An infinite loop (and actually any program) in Python can be terminated by pressing **CTRL + C**

Breaking out of the loop

- The execution of a loop can be ended by using the **break** keyword.

- Python ends the current loop execution **immediately** after **break** is encountered

- Note, that in most cases this is not necessary, as the condition for stopping the execution can be applied directly to the **while** statement.

Example

- Find a letter in a string:

```
string = raw_input("Enter a string:")
char = raw_input("Enter a character:")

index = 0
while index < len(string):
    if (string[index] == char):
        print "Found it!"
        break
    index = index + 1
```

Example (without the break):

```
string = raw_input("Enter a string:")
char = raw_input("Enter a character:")

index = 0
found = False
while index < len(string) and not found:
    if (string[index] == char):
        print "Found it!"
        found = True
    else:
        index = index + 1
```

Breaking out of the loop (cont.)

- Note, that only the execution of the current loop is terminated. If there are other loops surrounding it, their execution may still continue.

```
while True: # this goes on forever...
  print "hey"
  while True:
     # ...and this loop terminates in
     # each round.
     print "hello"
     break
```

Continue keyword

- Instead of breaking out of the loop, you can tell Python to continue the execution from the condition evaluation by using the **continue** keyword

- Again, in most cases the use of **continue** can be avoided by e.g. using the **if** statement.

- Note, that it is easy to create an infinite loop by using **continue**…

Example

- Try to calculate the roots of even numbers between 1 and 14. Does it work?

```
from math import sqrt
index = 0
while index < 14:
 if index % 2 != 0:
    continue
 else:
    print sqrt(index)
 index = index + 1
```

for loop in Python

- Python's **for** loop is sligthly different to for constructs in other languages

- Most commonly the **for** loop in Python is used to *iterate* through a sequence

- However, by using the **for loop** with Python's **range** function, we can easily iterate through a range of values.

The range function

- The **range** function returns a sequence of numbers in a **list**.

- The syntax is

```
    range(start, end, step)
```

- Again, the start value is inclusive, while the end value is exclusive.

The range function(cont.)

- If the step is omitted, Python uses default value of one.

- Examples:

`range(1,10)` à [1,2,3,4,5,6,7,8,9]

`range(2,8,2)` à [2, 4, 6]

`range(10,1,-1)` à [10, 9, 8, 7, 6, 5, 4, 3, 2]

The range function (cont.)

- With a single parameter, Python assumes a range from 0 until parameter:

- Examples:

  `range(10)` à [0,1,2,3,4,5,6,7,8,9]

  `range(2)` à [0,1]

The range function (cont.)

- This is convenient, as a range with N values can be created with a call range(N):

`range(10)` à [0,1,2,3,4,5,6,7,8,9]

`range(4)` à [0,1,2,3]

`range(1)` à [0]

Iterating through sequence

- The **for** loop can be used to iterate through a sequence created with the range function.

- The syntax of this construct:

  for *<variable>* **in <*list*>*:***

- In each step of the loop, one value from the range is assigned to *variable*

Iterating through range

- The range can be created beforehand:

```
r = range(10)
for i in r:
   print i
```

- ...but usually it's handy to create the range within the for statement

```
for i in range(10):
   print i
```

for loop in Python (cont.)

- Example: output numbers 1…15:

```
for i in range(1,16):
    print i
```

Examples (cont.):

- Output numbers 15...1 in decreasing order:

```
for i in range(15,0,-1):
    print i
```

- Output characters at even indeces in string:

```
str = raw_input("Give a string: ")
for i in range(0, len(str), 2):
    print str[i]
```

Range is a list of integers

- Note, that the range can be only used to create a list of *integers*; if you need other values, some consideration is required.

- To output values 0, 0.1, 0.2, ... , 1, use e.g.

```
for i in range(0,11):
    print i / 10.0
```

Strings as ranges

- Curiously, a string in Python is considered to be a sequence of characters. We return to this, when discussing Python lists.

- However, to iterate through characters in a string easily:

```
for char in "abcdefghijklmnop":
   print char
```

Question

- What does the following program output?

```
i = 0
while i < 10:
   print i
   i = i + 1
   if i == 5:
      i = 10
```

Question (2)

- How about the following program?

```
for i in range(10):
   print i
   if i == 5:
      i = 10
```

Few words about iteration

- The for loop creates a special variable called **iterator**, which is used to hold the current index in the iterable sequence

- Changing the value of variable used in for statement does not affect the iterator position.

Creating large ranges

- Since the range function actually generates a list of values into memory, iterating through very large lists takes a lot of excessive memory.

- This can be avoided by using the **xrange** function instead.

xrange

- The xrange function does not generate a list of items, but instead creates a **generator**
- The generator creates the items one at a time when iterated through
- Note, that if you actually need a list of items, you must use range instead of xrange

xrange (2)

- xrange works exactly similarily when used with the for statement:

```
# calculate the sum of values 1…20
sum = 0
for i in xrange(1,21):
    sum = sum + i
print sum
```

Another example

- Program that queries the user for a number, and proceeds to output all positive even numbers smaller than that:

```
n = input("Give a number: ")
for i in xrange(2, n):
   if i % 2 == 0:
      print i
```