Iterations and library functions

1DV501 - Introduction to Programming

Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

The slides are available in Moodle

September 14, 2020

Today ...

- Iterations (`while` and `for`)
- Built-in and library functions
  - A few built-in functions
  - The `math` module
  - The `random` module

Reading instructions: Sections 5.1-5.5, 6.1-6.4, 6.6

Iterations

Iteration (or loop) ⇒ repeat the same sequence of statements multiple times.

- In Python: `while`- and `for`-statements
- Example: `while`

```
# Print 1 to 10 using while
n = 1
while n <= 10:
    print(n, end=' ')  # No line-break
    n += 1
print()    # Add line break
```

Output: 1 2 3 4 5 6 7 8 9 10

- Example: `for`

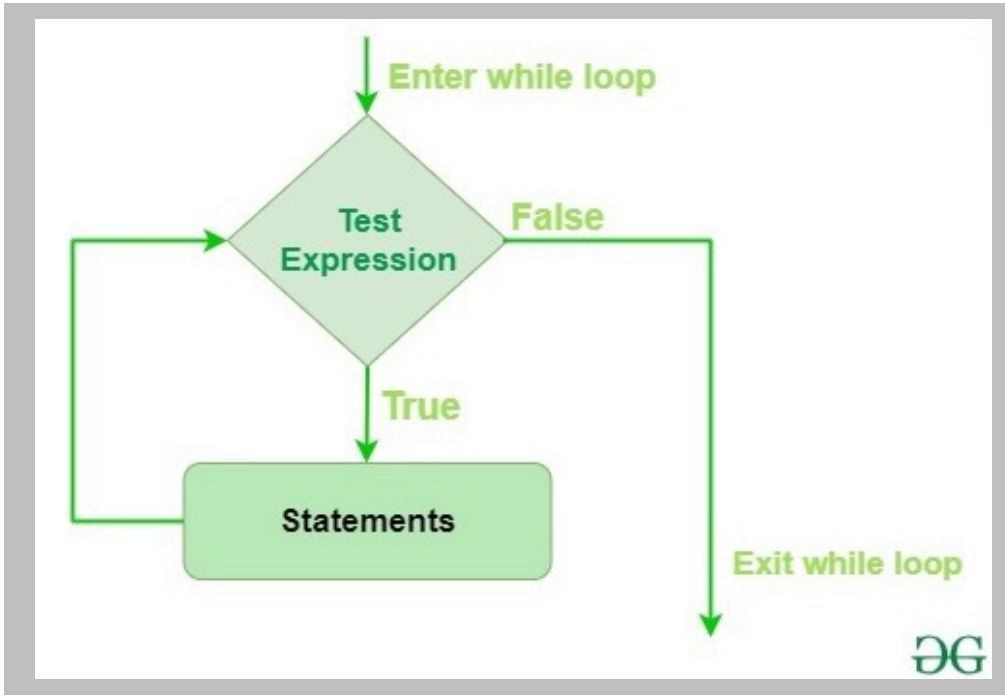
```
# Print 1 to 10 using for
for i in range(1,11):
    print(i, end=' ')  # No line-break
print() # Add line break
```

Output: 1 2 3 4 5 6 7 8 9 10

The `while` Statement

```
while "Test Condition":
    "Statements"
```

- The code `"Statements"` will be executed as long as `"Test Expression"` is `True`
- The code in `"Statements"` must be intended to be a part of the while statement
- `"Statements"` false ⇒ execution jumps to the code after the while statement



while Examples

```
# Print 1 to 10 using while
n = 1
while n <= 10:
    print(n, end=' ') # No break
    n += 1
print()     # Add line-break
```

Output: 1 2 3 4 5 6 7 8 9 10

```
# Find smallest N such
# that 1+2+3+...+N > 100
s = 0    # sum
N = 0
while s <= 100:
    N = N + 1  # N = 1,2,3,4, ...
    s = s + N  # s = 1,3,6,10, ...
print("Smallest N is", N)
```

- Repeat certain statements (the loop body) as long as a condition is false
- The 2nd example (Smallest N) shows when to use a `while` statement, when we don't know how many iterations that are needed but we know when to stop.
- The 1st example is better handled by a `for` statement (coming soon) since we know exactly how many iterations that are needed (10).

Nestled examples

Nestled ⇒ statements within statement

```
# Numbers dividable by 7 in range 1 to 100
n = 1
while n <= 100:
    if n % 7 == 0:  # Dividable by 7?
        print(n, end=' ')  # 7 14 21 ...
    n += 1
print()     # Add final line-break
```

```
# Do something while input is yes (y or Y)
entry = 'y'
while entry != 'N' and entry != 'n':
    entry = input("Enter Y to continue or N to quit: ")
    if entry == 'Y' or entry == 'y':
        print("Hello")     # Do something!
    elif entry != 'N' and entry != 'n':
        print(entry, "is not a valid input")
print("Done!")
```

Infinite loops

```
while True:
    print("Hello")
```

```
n = 1
str = ""     # empty string
while n < 10 or n > 0:
    n = n + 1
    str = str + "Hello"
```

- `while True:` $\Rightarrow$ loop never stops
- Q: What happens when executed?
- A: It just runs and runs ...
  (You stop it by Ctrl-C in the Terminal window.)

- A logical error
- `n < 10 or n > 0` is `True` for any `n`
- Program will crash since the string will get larger and larger and we will eventually run out of memory.

Infinite loops are often (but not always) a result of a logical error. They are sometimes useful when you want to do something (e.g. a sensor measuring the temperature) without ever stopping. They do not harm your computer in any way.

The `for` Statement

```
# Print 0,2,4,6,8,10
for i in range(0,11,2):
    print(i, end=' ')
print()
```

Output: 0 2 4 6 8 10

```
# Countdown from 10
for i in range(10,0,-1):
    print(i, end=' ')
print()
```

Output: 10 9 8 7 6 5 4 3 2 1

- `for i in range(0,11,2)` $\Rightarrow$ for each integer `i` in the range 0 to 10 using step size 2.
- **Notice:** The upper limit 11 is not included whereas the lower limit is.
- The variable `i` is called the *for counter*

The `range` function

The `range` function generates integer sequences and is rather powerful.
It comes in three versions:

- `range(stop)`: Considers by default the starting point as zero
- `range(start, stop)`: From `start` to `stop-1` with step size 1
- `range(start, stop, step)`: From `start` to `stop-1` with step size `step`

Examples

- `range(10)` ⇒ `0,1,2,3,4,5,6,7,8,9`
- `range(1, 10)` ⇒ `1,2,3,4,5,6,7,8,9`
- `range(1, 10, 2)` ⇒ `1,3,5,7,9`
- `range(2, 10, 2)` ⇒ `2,4,6,8`
- `range(10, 0, -2)` ⇒ `10,8,6,4,2`

Notice: Rather straight forward except that the `stop` value is not included.

The keywords `break` and `continue`

- `break` and `continue` are used to jump out of a loop at an arbitrary position.
- Example

```
while bool_expr:
  ...
  if bool_expr:
    break  # End the loop, jump to next_statement

  if bool_expr:
    continue # End this iteration, jump to while bool_expr:
  ...
}
next_statement
```

- `break` and `continue` are considered to make the code more difficult to understand ⇒ use them with care!

Is it a prime number? (Part 1)

$N > 1$ is a prime number $\Rightarrow$ Not dividable by any number in range 2 to $N - 1$

Problem: Write a program that checks if a given N is a prime number

Basic solution idea

- Error message if $N < 2$
- For each integer i in range 2 to $N - 1$
  - N dividable by i $\Rightarrow$ N is not a prime, interrupt loop
- Not dividable by any i $\Rightarrow$ N is a prime

Example runs

```
Enter an integer larger than 1: 47
47 is a prime number

Enter an integer larger than 1: 49
49 is NOT a prime number. It is dividable by  7

Enter an integer larger than 1: -7
Please follow the instructions!
```

Is it a prime number? (Part 2)

```
# Check if input is a prime number
n = int( input("Enter an integer larger than 1: "))

if n < 2:     # Must be larger than 1
    print("Please follow the instructions!")
else:
    prime = True
    for i in range(2,n):  # Check if prime
        if n % i == 0:
            prime = False
            break         # Jump from loop when not prime

    if prime:             # Present result
        print(n, "is a prime number")
    else:
         print(n, "is NOT a prime number. It is dividable by ",i)
```

Nestled Statements

```
# Print multiplication table
n = int( input("Please enter a positive integer: "))

if n < 1:
    print("Input must be positive!")
else:
    print("Multiplication table for ", n)
    for i in range(1,n+1):
        for j in range(1, n+1):
            print(i, " x ", j, " = ", i*j)
```

```
Please enter a positive integer: 3                    (Example run)
Multiplication table for  3
1  x  1  =  1
1  x  2  =  2
1  x  3  =  3
2  x  1  =  2
2  x  2  =  4
2  x  3  =  6
3  x  1  =  3
3  x  2  =  6
3  x  3  =  9
```

Problem solving with if, while, and for

- Understanding each control statement by itself is rather easy
- Solving problem requiring only one such statement is also often rather easy
- However, many problems require multiple nestled control statements
- Solution with nestled statements $\Rightarrow$ much harder $\Rightarrow$ much training needed
- We have a large number of such problems in Assignment 2

Assignment 2 is time consuming $\Rightarrow$ Get started!

Example: Count A

Write a program `count_A.py` that reads a string from the keyboard and then prints how many 'a' and 'A' the string contains. An example of what an execution might look like:

```
Provide a line of text: All cars got the highest safety grading A.
Number of 'a': 3
Number of 'A': 2
```

Sketch of a Solution

1. Read a line of text ⇒ a string `text`
2. For each character `c` in `text`
   - if `c = 'A'` ⇒ increase counter `nA` by 1
   - else if `c = 'a'` ⇒ ⇒ increase counter `na` by 1
3. Print result ⇒ Print `nA` and `na`

Hint: I should have waited with 1 (read line of text) until the end. Why?

Solution - count_A.py

```
# Count number of 'A' and 'a' in a string
text = input("Please provide a line of text: ")

na, nA = 0, 0
for c in text:
    if c == 'a':
        na += 1
    elif c == 'A':
        nA += 1
print("\nNumber of 'a': ", na)
print("Number of 'A': ", nA)
```

Notice: Iterating over all characters in a string is simple using a for statement

```
text = "This is a string"
for c in text:
    "Do something with character c"
```

A 10 minute break?

Using Functions

```
import random

# Print random numbers
n = int ( input("Number of random numers: "))

print(n, "random numbers: ", end=" ")
for i in range(n):          # n iterations
    r = random.randint(1,1000)
    print(r, end=" ")
print()
```

- The program above uses 5 different functions
- Built-in functions: `input()`, `int()`, `print()`, `range()`
  These functions are always available
- Library functions: `randint()`
  `randint` belongs to the module `random`

We will present a number of functions that might me useful in your assignments in the following slides. Both built-in functions (always available) and library functions (requires import).

Built-in Functions

Built-in functions are always available ⇒ no import needed

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

About a third of the functions above will be presented and used in this course

Common Built-in Functions

```
a, b, c, = 1.0, -2.2, 3.14

print("Values:",a, b, c, sep=", ")    # 1.0, -2.2, 3.14
print("Maximum:", max(a,b,c))         # 3.14
print("Minimum:", min(a,b,c))         # -2.2
print("Absolute value:", abs(b))      # 2.2
print("Round off:", round(c))         # 3
print("Hello", len("Hello"))          # Hello 5
print(type(a), type("Hello"))         # <class 'float'> <class 'str'>
```

- `max()`, `min()`, `abs()`, `round()` ⇒ see example above
- `float()`, `int()`, `bool()`, `str()` ⇒ type conversion
- `len("Hello")` ⇒ length of something. In this case a string
- `type()` ⇒ current type of variable or expression

The `math` Module

```
import math
pi = math.pi                        # Pi as a float

print(pi)                           # 3.141592653589793
print( math.degrees(pi/3) )         # 60
print( math.cos(pi/3) )             # 0.5

print( math.sqrt(2) )                # 1.4142135623730951
print( math.pow(4,3) )               # 64.0
print( math.floor(pi), math.ceil(pi) )          # 3 4
print( math.gcd(15,20) )             # 5
```

- `math` is an external module ⇒ must be imported
- Trigonometric functions: sin(), cos(), tan(), asin(), acos(), atan(), ...
  They work with radians by default
- `math.degrees( pi/3)` ⇒ convert radians to degrees
- `sqrt(x), pow(x,p)` ⇒ square root of x, x raised to the power of p
- `floor(x), ceil(x)` ⇒ Rounds off x downwards/upwards to the nearest integer
- `gcd(n,p)` ⇒ Greatest common devisor

Import modules and functions

Three equivalent import approaches

```
import math                     # Make math module available

print( math.sqrt(2) )           # Must make reference to math module
print( math.gcd(10,15) )
```

```
from math import sqrt, gcd      # Make sqrt and gcd from math available

print( sqrt(2) )                # No math reference required
print( gcd(10,15) )
```

```
from math import *              # Make all functions in math available

print( sqrt(2) )                # No math reference required
print( gcd(10,15) )
```

Use 1st approach when multiple math functions are needed.
Use 2nd approach when only 1-2 functions are needed
Avoid 3rd approach since name collisions are more likely

Short names for imported modules

```
import random                          # Make random module available

print( random.randint(0,100) )         # Random int in [0,100]
print( random.uniform(20,30) )         # Random float in [20,30]
```

```
import random as rd                    # Give random the name "rd"

print( rd.randint(0,100) )             # Use "rd" to reference random
print( rd.uniform(20,30) )
```

Use 1st approach when module name is short (e.g. `math`)
Use 2nd approach when module name is lengthy (e.g. `matplotlib`)

Terminate execution using `exit()`

Handle initial check using `if-else`

```
n = int( input("Enter a positive integer: ") )

if n < 1:
    print("Input must be positive")
else:
    "Do something with n ... ==> main part of program"
```

Handle initial check using `if` and `sys.exit()`

```
import sys

n = int( input("Enter a positive integer: ") )

if n < 1:
    print("Input must be positive")
    sys.exit().    # Terminates program execution

"Do something with n ... ==> main part of program."
```

Advantage: Avoid having to indent main part of program. Disadvantage: `exit()` terminates program prematurely $\Rightarrow$ sometimes a bit harder to understand.

Example: Stupid Encryption

A very simple (stupid?) way to encrypt a text would be to just shift each letter one step in the alphabet. That is, replace all letters in the text with the next letter in the alphabet

```
a --> b, b --> c,    ...    , y --> z, z --> a
A --> B, B --> C,    ...    , Y --> Z, Z --> A
```

All non-letters, for example digits, ? ,!, %, and whitespace, are left unchanged.

Exercise: Write a program `stupidencryption.py` that reads a line of text from the user and presents an encrypted version of the text according to the encryption method outlined above. An execution might look like this:

```
Provide a line of text: Was it a rat I saw?
Encrypted Text: Xbt ju b sbu J tbx?
```

Hints: A-Z have ASCII codes in range [65,90], a-z are in range [97,122], built-in function `ord()` gives the ASCII code for a character, and function `chr()` gives the character for an ASCII.

Basic idea: Convert each letter to ASCII, add 1, and convert back to character

stupidencryption.py

```
# str = "abcdefghijklmnopqrstuvwxyz 123 ABCDEFGHIJKLMNOPQRSTUVWXYZ.;?"
str = input("Provide a line of text: ")

result = ""
for c in str:
    asc = ord(c)   # ASCII code
    if c == 'z':   # z --> a
        result += 'a'
    elif c == 'Z': # Z --> A
        result += 'A'
    elif 65 <= asc <= 90:  # Upper case
        result += chr(asc+1)
    elif 97 <= asc <= 122: # Lower case
        result += chr(asc+1)
    else:                  # Handle non-characters
        result += c
print("Encrypted text:", result)
```

Programming: Old Java-test Exercise

Write a Java program `Square.java` that first reads any integer (higher than or equal to 3) from the keyboard and then prints a non-filled square of the type presented below. An execution might look like this:

```
Provide an integer 3 or higher: 5
The square for number 5
 *****
 *   *
 *   *
 *   *
 *****
```

An error message should be given, and the program should terminate, if the user-provided integer value is below three.

Let us solve the problem in Python!

Solution - square.py

```
sz = int ( input("Enter an integer 3 or higher: "))

if sz < 3:  # Check input
    print("The size must 3 or higher")
else:
    # Two types of square lines
    stars = ""
    for i in range(sz):
        stars += "*"

    line = "*"
    for i in range(sz-2):
        line += " "
    line += "*"

    # Print square
    print("\nThe square for number", sz)
    print(stars)
    for i in range(sz-2):
        print(line)
    print(stars)
```