# Bioinformatics

## Python Programming
## Introduction to the Python Language

---

Sérgio Crisóstomo

2026

Departamento de Ciência de Computadores

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

# Contents

# More about loops

# Break and continue in a loop

Two statements that allow you to alter the execution of a loop:

| | |
|---|---|
| **break** | exit the loop in the middle |
| **continue** | skip to the next iteration |

Additionally:

| | |
|---|---|
| **return** | terminates the function and returns a result (also interrupts any loop) |

# Break with a for loop

```
for i in [12, 16, 17, 24, 29]:
    if i % 2 == 1:  # if it is odd
        break          # exit the loop
    print(i)
print("end")
```

Result:

```
12
16
end
```

# Continue with a for loop

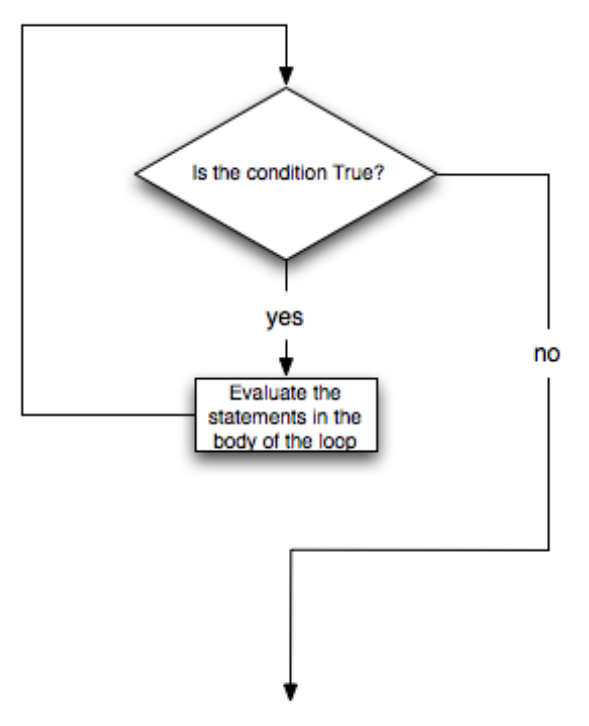
```
for i in [12, 16, 17, 24, 29, 30]:
    if i % 2 == 1:      # if it is odd
        continue          # skip to the next
    print(i)
print("end")
```

Result:

```
12
16
24
30
end
```

# Exiting a while loop

The test for the *while* loop occurs before the body is executed.




We can use *break* to place a test inside the loop body.

# Example: test in the middle of the body

```
total = 0
while True:
    answer = input("Enter a number (or leave blank)")
    if answer == '':
        break           # exit the loop
    total += int(answer)
print("Total = ", total)
```

# Example: test at the end of the body

```
import random
# choose an integer between [1, 1000]
number = random.randint(1, 1000)
guesses = 0
while True:
    guess = int(input("Number between 1 and 1000: "))
    guesses += 1
    if guess > number:
        print(guess, "is too high.")
    elif guess < number:
        print(guess, "is too low.")
    else:
        break
print(guess, "is correct.")
print(guesses, "attempts.")
```

# Strings

# Strings

- They are sequences of characters
- We can treat them as a single entity
- But we can also access individual characters

# Example

```
>>> txt = 'Banana'
>>> txt[0]
'B'
>>> txt[1]
'a'
>>> txt[2]
'n'
>>> len(txt)
6
```

# Indexes


- Indices from `0` to `len(txt)-1`
- Characters: `txt[0]`, `txt[1]`, ...
- Last character: `txt[len(txt)-1]`
- Second to last character: `txt[len(txt)-2]`
- Negative indexes count **from the end to the beginning**:

```
txt[-1] == txt[len(txt)-1]
txt[-2] == txt[len(txt)-2]
```

# Negative indexes

```
>>> txt = 'Banana'
>>> txt[5]
'a'
>>> txt[-1]
'a'
>>> txt[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
>>> txt[-6]
'B'
```

# Slices

**`txt[i:j]`** sub-string between indices $i$ and $j-1$ inclusive

**`txt[i:]`** sub-string from index $i$ to the end

**`txt[:j]`** sub-string from the beginning to index $j-1$ inclusive

```
>>> txt = 'Banana'
>>> txt[:3]
'Ban'
>>> txt[3:]
'ana'
>>> txt[2:5]
'nan'
```

# Strings are immutable

- We cannot modify characters in a string
- But we can build a new string from others

# Example

```
>>> txt = 'Bamana'
>>> txt[2] = 'n'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment

>>> new = txt[:2] + 'N' + txt[3:]
>>> new
'BaNana'

>>> txt = txt[:2] + 'n' + txt[3:]
>>> txt
'Banana'
```

# String comparison

***txt1* <= *txt2*** less than or equal to by lexicographical order (i.e., dictionary order)

```
>>> 'abba' <= 'banana'
True
>>> 'abbot' <= 'abacaxi'
False
>>> 'B' <= 'a'
True
>>> print(ord('B'), ord('a'))
66 97
```

The order between characters corresponds to the order of their *numeric codes* (Unicode system).

# Test occurrence

***txt1* in *txt2*** tests if *txt1* occurs within *txt2*

```
>>> 'ana' in 'Banana'
True

>>> 'mana' in 'Banana'
False
```

# Iterating over a string (1)

Using a loop over all valid indices:

```
for i in range(len(txt)):
    # i = 0, 1, ..., len(txt)-1
    print(txt[i])
```

# Iterating over a string (2)

Using a loop directly over the characters in the string:

```
for ch in txt:
    # ch = txt[0], txt[1], ..., txt[len(txt)-1]
    print(ch)
```

- Avoids the need for an index variable
- More generally: the `for` loop allows iteration over any *sequences*

# Searching for the first occurrence

Let's write a function `first(ch,txt)` that:

- searches if a character `ch` occurs in the string `txt`;
- if true, returns the *index* of the first occurrence;
- if false, returns -1.

# Example

```
>>> first('n', 'banana')
2
>>> first('m', 'banana')
-1
```

# Searching for the first occurrence

```
def first(ch:str, txt:str) -> int:
    "Search for the first occurrence of c in txt."
    for i in range(len(txt)):
        if txt[i] == ch: # found it
            return i     # ends and returns the index
    # end of the loop
    return -1   # not found: return -1
```

# String methods

- Strings support several *predefined operations*
- We use the *method invocation* syntax:

$$txt.operation(arg_1, arg_2, \ldots)$$

- Later: we will see examples of other *objects* and their respective *methods*

# String methods (cont.)

| | |
|---|---|
| **find** | index of the first occurrence |
| **replace** | replace occurrences |
| **upper** | convert lowercase letters to uppercase |
| **lower** | convert uppercase letters to lowercase |

Examples:

```
>>> txt = "Banana"
>>> txt.upper()
'BANANA'
>>> txt.find('ana')
1
>>> txt.replace('a','A')
'BAnAnA'
```

# More operations on strings

Let `txt` be a string.

**`txt.split()`** split the string into a list of parts delimited by spaces

**`txt.split(sep)`** split the string into a list of parts delimited by the string *sep*

**`txt.join(seq)`** join a sequence of strings into one using *txt* as a separator

**`txt.lower()`** return a copy of the string converted to lowercase

**`txt.upper()`** return a copy of the string converted to uppercase

**`txt.capitalize()`** return a copy of the string with its first character capitalized

**`txt.strip()`** return a copy of the string with leading and trailing whitespace removed

# More operations on strings (cont.)

**`txt.strip(chars)`** return a copy of the string with leading and trailing *chars* removed

**`txt.replace(old, new)`** return a copy of the string with all occurrences of *old* replaced by *new*

**`txt.find(sub)`** return the lowest index in the string where substring *sub* is found (or -1 if not found)

**`txt.count(sub)`** return the number of non-overlapping occurrences of substring *sub*

**`txt.startswith(prefix)`** return `True` if the string starts with the specified *prefix*

**`txt.endswith(suffix)`** return `True` if the string ends with the specified *suffix*

**`txt.isdigit()`** return `True` if all characters in the string are digits

# More operations on strings (cont.)

**`txt.isalpha()`** return `True` if the string contains only alphabetic letters

**`txt.isdigit()`** return `True` if the string contains only digits

**`txt.isalnum()`** return `True` if the string contains only alphanumeric characters (letters and digits)

**`txt.isspace()`** return `True` if the string contains only whitespace characters

**`txt.islower()`** return `True` if all cased characters in the string are lowercase

**`txt.isupper()`** return `True` if all cased characters in the string are uppercase

**`txt.istitle()`** return `True` if the string is title-cased (words start with uppercase)

# Examples

```
>>> "the weapons and the barons".split()
['the', 'weapons', 'and', 'the', 'barons']

>>> "abc-de-fgh".split('-')
['abc', 'de', 'fgh']

>>> " ".join(['the', 'weapons', 'and', 'the', 'barons'])
'the weapons and the barons'

>>> "--".join(['the', 'weapons', 'and', 'the', 'barons'])
'the--weapons--and--the--barons'
```

# Lists

# Lists

- Ordered sequences of elements
- Can contain elements of any type
- Possibly with repetitions
- Elements are identified by indices

# Example

```
>>> foods = ["bread", "bread", "cheese", "cheese"]
>>> foods[0]
'bread'
>>> foods[1]
'bread'
>>> foods[2]
'cheese'
>>> len(foods)
4
```

# Lists by extension

- List with *n* elements: `[e1, e2, ..., en]`
- Order matters
- Repeated elements may occur
- Can be an empty list: `[]`

# Accessing elements

- Indexing operator: `list[i]`
- Indices between 0 and `len(list)-1`
- Negative indices: access from the end
- Invalid indices give an runtime error

# Slices

| | |
|---|---|
| `list[i:j]` | elements between $i$ and $j-1$ inclusive |
| `list[i:]` | elements from $i$ to the end |
| `list[:j]` | elements from the first to $j-1$ inclusive |
| `list[:]` | all elements (copy of the list) |

```
>>> vowels = ['a','e','i','o','u']
>>> vowels[1:4]
['e', 'i', 'o']
>>> vowels[:3]
['a','e','i']
>>> vowels[3:]
['o','u']
>>> vowels[:]
['a', 'e', 'i', 'o', 'u']
```

# Slices (cont.)

### General form

`list[i:j:k]` elements from $i$ to $j - 1$ with increments of $k$

Negative increments: traverse the list backwards.

```
>>> vowels[::2]        # even indices
['a','i','u']
>>> vowels[1::2]       # odd indices
['e','o']
>>> vowels[::-1]       # reverse the list
['u', 'o', 'i', 'e', 'a']
```

# Iterating over indices and elements

```
for i in range(len(list)):
    print(i, list[i])
```

- Loop over indices $i$ from 0 to $len(list) - 1$
- Element $list[i]$ associated with index $i$

# Iterating over all elements

```
for value in list:
    print(value)
```

- Avoids the need to explicitly manage the index
- Preferable when you need the values but not the indices

# List operations

`+` concatenation

*n*`*` repetition (*n* times)

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a + b
[1, 2, 3, 4, 5, 6]
>>> 3*a
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

# Lists are mutable

We can modify or add elements:

```
>>> beatles = [1, 2, 3]
>>> beatles[0] = "john"
>>> beatles[2] = "ringo"
>>> beatles
['john', 2, 'ringo']

>>> beatles[1:2] = ['paul', 'george']
>>> beatles
['john', 'paul', 'george', 'ringo']
```

# Removing elements from a list

```
>>> beatles = ['john', 'paul', 'george', 'ringo']
>>> del beatles[0]
>>> beatles
['paul', 'george', 'ringo']
```

Alternative:

```
>>> beatles = ['john', 'paul', 'george', 'ringo']
>>> beatles[0:1] = []
>>> beatles
['paul', 'george', 'ringo']
```

# Names and objects

It is important to distinguish the name of a variable from the collection of values associated with that name.

# Names and objects (1)

Two names, two separate lists:

```
>>> a = [1,2,3]
>>> b = [1,2,3]
>>> a[0] = 'oops'
>>> print(a, b)
['oops', 2, 3] [1, 2, 3]
```

# Names and objects (2)

Two names, only one list:

```
>>> a = [1,2,3]
>>> b = a
>>> a[0] = 'oops'
>>> print(a, b)
['oops', 2, 3] ['oops', 2, 3]
```

# Names and objects (3)

Two names, two lists (by making a copy):

```
>>> a = [1,2,3]
>>> b = a[:]
>>> a[0] = 'oops'
>>> print(a, b)
['oops', 2, 3] [1, 2, 3]
```

# List methods

Some predefined methods:

| | |
|---|---|
| **append** | add an element to the end |
| **insert** | add an element at a position |
| **extend** | append all elements from another iterable (e.g., another list) |
| **remove** | remove the first occurrence of an element |
| **pop** | remove and return the element at a given index (default is the last element) |
| **clear** | remove all elements from the list |
| **index** | return the index of the first occurrence of an element |
| **count** | return the number of times an element appears in the list |

# List methods (cont.)

| | |
|---|---|
| **sort** | sort the elements in ascending order |
| **reverse** | reverse the order of the elements in place |
| **copy** | return a shallow copy of the list |

- Usage: `list.method(arguments)`
- Modify the list (most of these operate in-place)

# Examples

```
>>> beatles = ['john','paul']
>>> beatles.append('george')
>>> beatles.append('ringo')
>>> beatles
['john', 'paul', 'george', 'ringo']
>>> beatles.insert(0, 'paul')
>>> beatles
['paul', 'john', 'paul', 'george', 'ringo']
>>> beatles.sort()
>>> beatles
['george', 'john', 'paul', 'paul', 'ringo']
```

For more information:

```
>>> help(list)
```

# Lists within lists

- Lists can contain other lists
- This way, we can represent tables or matrices

```
>>> matrix = [[1,2,-1],
              [3,1,0],
              [0,1,-2]]
>>> matrix[1][0]
3
>>> matrix[1][0] = -3
>>> matrix
[[1, 2, -1], [-3, 1, 0], [0, 1, -2]]
```

# Tuples

# Tuples

- Ordered sequences of elements:

```
(e1, e2, ..., en)
```

- Access to elements by indices
- Unlike lists, tuples are immutable

# Example

```
>>> grade = ('Pedro', 12)
>>> grade[0]
'Pedro'
>>> grade[1]
12
>>> grade[0] = 'Joao'
TypeError: 'tuple' object does not support
item assignment
```

# Operations on tuples

Operators `+` and `*` similar to those for lists:

```
>>> t1 = ('Pedro', 12)
>>> t2 = ('Joao', 14)
>>> t1 + t2
('Pedro', 12, 'Joao', 14)
>>> 3*t1
('Pedro', 12, 'Pedro', 12, 'Pedro', 12)
```

# Assigning to tuple variables

```
>>> (x,y) = (5,-7)
>>> x
5
>>> y
-7
```

Or simply:

```
>>> x,y = 5,-7
>>> x
5
>>> y
-7
```

# Lists and tuples combined

Let's represent an address book as a list of pairs *name*/*email*:

```
[('Pedro Vasconcelos', 'pbv@dcc.fc.up.pt'),
 ('Pedro Brandão', 'pbrandao@dcc.fc.up.pt'),
 ('João Pedro Pedroso', 'jpp@dcc.fc.up.pt')]
```

Operations:

- add an entry (name and email)
- search email by name

# Add an entry

```
def add(address_book, name, email):
    "Add a name and email to the address book."
    address_book.append((name,email))
```

# Search for a name (1)

```
def search(address_book, txt):
    "Search emails by part of the name."
    emails = []
    for pair in address_book:
        if txt in pair[0]:          # txt occurs in the name?
            emails.append(pair[1]) # add email
    return emails
```

# Search for a name (2)

```
def search(address_book, txt):
    "Search emails by part of the name."
    emails = []
    for (name,email) in address_book:
        if txt in name:              # txt occurs in the name?
            emails.append(email)     # add email
    return emails
```

# Examples

```
>>> address_book = []
>>> add(address_book, "Pedro Vasconcelos",
                  "pbv@dcc.fc.up.pt")
>>> add(address_book,"João Pedro",
                 "jpp@dcc.fc.up.pt")

>>> search(address_book, "João")
['jpp@dcc.fc.up.pt']

>>> search(address_book, "Pedro")
['pbv@dcc.fc.up.pt', 'jpp@dcc.fc.up.pt']
```

# More on lists processing

# Aggregations

An *aggregation* is an operation that converts a list of values into a single result.

Examples:

- **counting** the number of values
- **summing** all the values
- determining the **maximum** value
- determining the **minimum** value
- determining the **average** of the values

# Predefined aggregations

Some aggregations are available as predefined functions in Python:

- `len()` count values (length)
- `sum()` sum all the values
- `max()` determine the maximum
- `min()` determine the minimum

# Examples

```
>>> values = [0.5, 1.5, 0.5]
>>> len(values)
3
>>> sum(values)
2.5
>>> max(values)
1.5
>>> min(values)
0.5
```

# Other aggregations

Let's define functions to calculate other aggregations.

Examples:

- the arithmetic **average** of values
- the **product** of the values

# Arithmetic mean of a sequence

```
def mean(values: List[float]) -> float:
    "Calculate the arithmetic mean of a list."
    return sum(values)/len(values)

>>> values = [0.5, 1.5, 0.5]
>>> mean(values)
0.8333333333333334
```

Note that there is no mean for the empty list:

```
>>> mean([])
ZeroDivisionError: ...
```

# Product of values

$$\text{product of } [v_1, v_2, \ldots, v_n] = v_1 * v_2 * \cdots * v_n$$

Algorithm:

1. a loop over the list of values
2. accumulate the *partial product*
3. the accumulator should be initialized with 1
   (neutral element of multiplication)

# Product of values

```
def product(values: List[float]) -> float:
    "Product of values in a list."
    acum = 1.0  # accumulator for the product
    # iterate over all values
    for x in values:
        acum = acum*x
    # end of loop; return the result
    return acum
```

# Examples

```
>>> product([2.0, 3.0, 4.0])
24.0
>>> product(range(1,5))
24.0
>>> product([])
1.0
```

# Removing duplicate elements

We are going to define a function to **remove duplicate elements** from a list.

Example: if the original list is

```
[2, 1, 3, 4, 1, 4, 1]
```

then the result should be the list

```
[2, 1, 3, 4]
```

where each value occurs only once, in the same order as the original list.

# Removing duplicates: Algorithm 1

Let *xs* be the original list:

1. We construct a new list *ys*
2. Initially, $ys = [\,]$
3. For each element *x* in the given list:
   if *x* does not yet occur in *ys*, then we add *x* to the list *ys*

# Removing duplicates: Implementation 1

```
def remove_duplicates1(lst: List[Any]) -> List[Any]:
    "Builds a new list without duplicates."
    # a new list (initially empty)
    new_list: List[Any] = []
    # iterate over the elements of the given list
    for x in lst:
        # if x has not occurred yet
        if not (x in new_list):
            # add it to the new list
            new_list.append(x)
    # return the list without duplicates
    return new_list
```

# Examples

```
>>> remove_duplicates1([2,1,3,3,1,4,1])
[2, 1, 3, 4]

>>> remove_duplicates1([2,2,2])
[2]

>>> remove_duplicates1([1,2,3])
[1, 2, 3]
```

# Files

# Files

- A coherent aggregation of information, e.g.:
  - a text document;
  - the source code of a Python program;
  - an image captured by a digital camera.
- Identified by a *path* in the file system
- Unlike program variables: files are persistent
- Physical media: magnetic disks, SSDs, flash memory, . . .

# Working with files

Three steps:

1. open the file
2. read and/or write to the file
3. close the file

# Opening a file

```
f = open(path, mode)
```

Modes:

| | |
|---|---|
| `'r'` | read (file must already exist) |
| `'w'` | write (if the file already exists: removes the content) |
| `'a'` | write (if the file already exists: appends to the end) |
| `'w+'` | read and write |

Important: you should always use `close` to ensure that the file is written correctly!

The result of `open` is an object of type `file`:

```
>>> f = open("test.dat","w")
>>> f
<_io.TextIOWrapper name='test.dat' mode='w'
 encoding='UTF-8'>
>>> type(f)
<class '_io.TextIOWrapper'>
```

We use methods to operate on the file.

# Writing example

The program

test.py

```
f = open("test.dat", "w")
f.write("Hello world!\n")
f.write("Goodbye world...\n")
f.close()
```

produces the following file:

test.dat

```
Hello world!
Goodbye world...
```

# Reading example

```
>>> f = open("test.dat", "r")
>>> txt = f.read()
>>> txt
'Hello world!\nGoodbye world...\n'
>>> f.close()
```

# If the file does not exist

```
>>> f = open("test.cat", "r")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IOError: [Errno 2] No such file or directory:
  'test.cat'
```

# Text or binary files?

**Text files** sequences of character codes (e.g. ASCII, ISO-LATIN-1, UTF-8)

**Binary files** specialized formats (e.g. JPEG for images, MP3 for audio, ELF for executables)

- By default: `open` assumes the file is text
- In this course, we will only use text files

# Text encoding

- The most common text encoding system on the *web* and on Linux systems is UTF-8
  - it supports text with characters from all European languages, Arabic, Hebrew, and others
- On Windows: the text encoding may not be UTF-8 by default
- To ensure that we open files using UTF-8 we can specify the encoding explicitly:

```
f = open(path, mode, encoding="utf-8")
```

# Searching for word occurrences

A function to search for a word in a file:

- prints the number and line where the word occurs
- simplified version of the `grep` command from UNIX

# Usage example

```
>>> search("Cupid", "sonnets.txt")
2595:Cupid laid by his brand and fell asleep:
2608:    Where Cupid got new fire; my mistress' eyes.
```

# Searching for word occurrences

```
def search(word, file):
    "Occurrences of a word in a file."
    f = open(file, "r", encoding="utf-8")
    n = 1          # line counter
    while True: # repeat
        line = f.readline()  # a new line
        if line == "":      # end of file?
            break
        if word in line:  # does the word occur?
            print("%d:%s" % (n,line))
        n = n + 1            # another line
    # end of loop
    f.close()
```

# Text formatting

# Text formatting

Sometimes we need to specify how the results of a computation are displayed:

- the number of digits and decimal places;
- with or without leading/trailing zeros;
- columns aligned in a table.

Programming languages allow us to do this with commands or libraries for text formatting.

# Text formatting (cont.)

- There are several ways to format in Python
- We will see how to use `%` as a formatting operator
- Compatible with both Python 2 and 3

# Formatting operator

*format* `%` *values*

- the *format* is a string with fields marked by `%` characters
- one or more *values* aggregated in a tuple
- the *result* is a string with the fields replaced by the corresponding values

Examples:

```
>>> "The value of Pi is %f" % math.pi
'The value of Pi is 3.141593'
>>> "%d/%d/%d" % (1, 6, 2013)
'1/6/2013'
```

# Some format fields

**%d** signed integer

```
"%d/%3d/%-3d" % (5, 5, 5)
'5/  5/5  '
```

**%e, %f, %g** floating point, exponential or decimal format

```
"%f %.3f %e" % (math.pi,math.pi,math.pi)
'3.141593 3.142 3.141593e+00'
```

**%s** string

```
"(%s/%4s/%-4s)" % ("A", "BC", "D")
'(A/  BC/D   )'
```

**%%** the % character

```
"%d%%" % 12
12%
```

# Example

Print a table of sine and cosine functions in the range $[0, 2\pi]$.

First version (without formatting).

```
from math import *
print("x", "sin(x)", "cos(x)")
for i in range(11):
    x = 2*pi/10 * i
    print(x, sin(x), cos(x))
```

# Example (cont.)

Result:

```
x   sin(x)   cos(x)
0.0 0.0 1.0
0.6283185307179586 0.587785252292473 0.8090169943749475
1.2566370614359172 0.9510565162951535 0.3090169943749475
1.884955592153876 0.9510565162951536 -0.3090169943749473
2.5132741228718345 0.5877852522924732 -0.8090169943749473
3.141592653589793 1.2246467991473532e-16 -1.0
3.7699111843077517 -0.587785252292473 -0.8090169943749476
4.39822971502571 -0.9510565162951535 -0.3090169943749476
5.026548245743669 -0.9510565162951536 0.3090169943749473
5.654866776461628 -0.5877852522924734 0.8090169943749473
6.283185307179586 -2.4492935982947064e-16 1.0
```

# Example (cont.)

Second version (using formatting):

```
print('%7s  %7s  %7s' % ('x', 'sin(x)', 'cos(x)'))
for i in range(11):
    x = 2*pi/10 * i
    print('%7.4f  %7.4f  %7.4f' % (x,sin(x),cos(x)))
```

Legend:

- **`%7.4f`** floating-point field with 7 characters total and 4 decimal places;
- **`%7s`** text field with 7 characters total.

# Example (cont.)

Result:

```
      x    sin(x)    cos(x)
 0.0000    0.0000    1.0000
 0.6283    0.5878    0.8090
 1.2566    0.9511    0.3090
 1.8850    0.9511   -0.3090
 2.5133    0.5878   -0.8090
 3.1416    0.0000   -1.0000
 3.7699   -0.5878   -0.8090
 4.3982   -0.9511   -0.3090
 5.0265   -0.9511    0.3090
 5.6549   -0.5878    0.8090
 6.2832   -0.0000    1.0000
```