

Control flow

Inteligencia Artificial en los Sistemas de Control Autónomo
Máster Universitario en Ingeniería Industrial

Departamento de Automática

Objectives

1. Understand control flow in Python
2. Understand functions and its syntax in Python
3. Design elemental algorithms
4. Implement elemental algorithms in Python

Table of Contents

1. Conditions and loops

- if Statements
- for Statements
- Branching statements
- pass Statements

2. Functions

- Defining functions
- Global and local variables
- Default argument values
- Keyword arguments

3. Coding conventions

- Documentation strings
- Coding style

4. Examples

- Example 1
- Example 2

Conditions and loops

if Statements (I)

Conditional statements implement decision making

- It is based on a condition
- The result is boolean
- Remember: Indentation defines the body code

```
temperature = float(input('What is the
    temperature? '))
if temperature > 70:
    print('Wear shorts. ')
else:
    print('Wear long pants. ')
print('Get some exercise outside. ')
```

Good practice: The usage of `else` is optional, try to avoid it!

Conditions and loops

if Statements (II)

Many times decisions are not binary (true/false): `elif`

- Conditions are evaluated until first true
- If all conditions are false, then it executes `else`
- `else` is optional (try not to use it!)

`elif` Statement

```
if [ condition1 ]:  
    # Some code here  
elif [ condition2 ]:  
    # Some other code  
elif [ condition3 ]:  
    # Some other code  
else:  
    # More code
```

Conditions and loops

if Statements (III)

Complex if Statement

```
x = int(input("Please enter an integer: "))

if x < 0:
    x = 0
    print('Negative changes to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')

print(x)
```

Conditions and loops

for Statements (I)

- Sometimes we have to repeat a task: Loops
 - Other languages iterate over a condition
 - For instance, in C: `for (i=0; i<10; i++)`
- Two loop statements in python: `while` and `for`
- In Python, `for` iterates over a sequence (lists or strings)
 - In each iteration, it assigns a sequence value to a variable

for Statement example

```
list = [ 'cat', 'window', 'dog' ]  
  
for x in list:  
    print(x)
```

for Statement example

```
string = "Hello word"  
  
for x in string:  
    print(x)
```

Conditions and loops

for Statements (II)

Sometimes, we need to iterate over a sequence of numbers

- `range(n)`: It returns a sequence $0, \dots, n - 1$

`range()` example

```
for i in range(5):  
    print(i)
```

Alternative notation

```
a = [ 'Mary ', 'had ', 'a ' ]  
  
for i in range(len(a)):  
    print(i, a[i])
```


Conditions and loops

Branching statements (I)

We do not always want to iterate over the loop

- `break`: Exit the loop
- `continue`: Jump to next iteration
- `break` and `continue` are valids in loops

break use

```
for i in foo:  
    # Some code  
    if i == 3:  
        break  
    # More code
```

continue use

```
for i in foo:  
    # Some code  
    if i == 3:  
        continue  
    # More code
```

Conditions and loops

Branching statements (II)

Break example

```
number = int(input('Enter a number: '))

if number > 1:
    is_prime = True
    for divider in range(2, number):
        if number % divider == 0:
            is_prime = False
            break
    else:
        is_prime = False

if is_prime:
    print('The number {0} is prime.' .format(number))
else:
    print('The number {0} not is prime.' .format(number))
```

Conditions and loops

Branching statements (III)

What this is doing?

```
for i in range(2, 10):  
    for x in range(2, i):  
        if i % x == 0:  
            print(i, 'equals ', x, '*', i//x)  
            break  
        else:  
            print(i, ' is prime number')
```

Conditions and loops

pass statements

`pass`: A statement that does nothing ...

- ... yes, nothing
- It is used to avoid compilation errors
- Code bodies that do nothing

Example 1

```
# Infinite loop
# waiting an
# interrupt

while True:
    pass
```

Example 2

```
# Empty class

class MyEmptyClass:
    pass
```

Example 3

```
def initlog(*args):
    # Ignore function
    pass
```

Functions

Defining functions (I)

Function: A piece of code that can be used several times

- Lazy programmers are good programmers
- Code reuse

Functions can be used with parameters

- Define a function before using it

Function 1

```
def printHello():  
    print("Hello")  
  
printHello()
```

Function 2

```
def printTwice(string):  
    print(string)  
    print(string)  
  
printTwice(string)
```

Hint: If you have to use code more than once, place it in a function

Functions

Defining functions (II)

A function may be as complex as needed

Fibonacci series function

```
#!/usr/bin/python

a, b = 0, 1 # Init variables

while b < 10: # This is a loop
    print("b = ", b)
    print("a = ", a) # Indentation is very important
    here!
    a, b = b, a+b
```

How it works? Example: Calculation of `fib(4)`

New Python elements:

- docstrings, for automatic documentation
- Keywords arguments

Functions

Defining functions (III)

Boring (albeit useful) fact: A function is just another variable

```
>>> fib
<function fib at 0x1006771e0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
>>> f
<function fib at 0x1006771e0>
```

Functions

Defining functions (IV)

Python functions can return values

Return Fibonacci series

```
def fib2(n):  
    """Print a Fibonacci series up to n """  
    result = [] # Declare a new list  
    a, b = 0, 1  
    while a < n:  
        result.append(a) # Add to the list  
        a, b = b, a+b  
    return result
```

New Python features

- The return statement
- Adding elements to a list

Functions

Defining functions (V)

Example:

Conversion of degrees

```
def fahrenheit_centigrados(x):
    """ Conversion de grados Farenheit a Centigrados """
    return (x - 32) * (5 / 9.0)

def centigrados_fahrenheit(x):
    """ Conversion de grados Centigrados a Farenheit """
    return (x * 1.8) + 32
```

Functions

Global and local variables (I)

Variable scope:

- **Global variables:** Defined outside of the functions.
 - Can be read within and outside the functions.
- **Local variables:** Defined within of a function, including formal parameters.
 - Invisibles outside the function.

Example

```
a = 5

def f():
    a = 2
    print(a) # 2
    return

f()
print(a) # 5
```

Functions

Global and local variables (II)

It is possible to modify the *global* object within a function?

Example 1

```
a = 5

def f():
    a = 2
    print(a) # 2
    return

f()
print(a) # 5
```

Example 2

```
a = 5

def f():
    global a
    a = a - 1
    return

f()
print(a)
```

Functions

Global and local variables (III)

Example 3

```
def increase(p):  
    p = p + 1  
    return p  
  
a = 1  
b = increase(a)  
  
print('a:', a)  
print('b:', b)
```

Functions

Global and local variables (IV)

To modify a global object in a function, it must be declared using the statement `global`.

use of `global` statement

```
a = 5

def f():
    global a
    a = 0
    print(a)
    return

f()
print(a)
```

Write-protection:

- The immutable variables (numbers, strings, tuples) → **yes**.
- The mutable variables (lists, dictionaries) → **no**.

Functions

Global and local variables (V)

Examples:

Example 1

```
lista = [ "Juan", "Pepe" ]

def f():
    lista.pop()

print( lista )
f()
print( lista )
```

Ejemplo 2

```
lista = [ "Juan", "Pepe" ]

def f():
    lista = [ "Maria" ]

print( lista )
f()
print( lista )
```

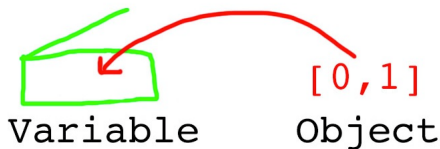
What will happen if the list `lista` is declared as global?

Functions

Global and local variables (VI)

Parameter passing in Python

- Python is **pass-by-object-reference**.
 - A variable and an object are different things.
 - A function receives a reference to (and will access) the same object in memory as used by the caller.
 - The function provides its own box and creates a new variable for itself.



Source

Functions

Global and local variables (VII)

Parameter passing in Python

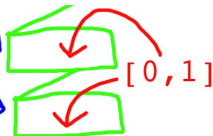
```
def append(list):
```

```
    list.append(1)
```

```
list = [0]
```

```
append(list)
```

```
print list
```



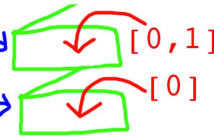
```
def reassign(list):
```

```
    list = [0, 1]
```

```
list = [0]
```

```
reassign(list)
```

```
print list
```



Want to know more? [Click here!](#)

Pass-by-object-reference

Object references are passed by value

Functions

Global and local variables (VIII)

Summary:

- **Global objects:** Objects defined outside the function.
- **Local objects:** Objects defined within the function.
- **Global objects** can always be read within a function.
- Modification of a global object, `object`, within a function:
 - If `object` is **immutable** → Use `global object` within the function.
 - If `object` is **mutable** →
 - If you want to change by an assignment → Use `global object` within the function.
 - If you want to change **using methods** → It is not necessary to use `global object` within the function.

Functions

Default argument values (I)

Python supports default arguments:

- Powerful and simple feature.
- Simpler (and more flexible) function calls.

```
def ask_ok(prompt, retries=4, complaint="Yes or no"):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
    print(complaint)
```

Functions

Default argument values (II)

New Python features

- The `in` keyword
- Exceptions (error handling)

The function can be invoked in several ways:

- `ask_ok('Do you really want to quit')`
- `ask_ok('OK to overwrite the file?', 2)`
- `ask_ok('OK to overwrite the file?', 2, 'Come on, yes or no!')`

Functions

Keyword arguments

Function arguments can be named:

- It overrides classic positional arguments.
- Order does not matter.
- Positional arguments must be first.

```
def foo(bar, baz):
    print(bar, baz)
```

```
foo(1, 2)
foo(baz = 2, bar = 1)
```

```
def foo(bar = "hello", baz = "bye"):
    print(bar, baz)
```

```
foo()
foo("hi")
foo(baz = "hi")
```

Arbitrary number of arguments:

- Arguments as `*arg1` and `**arg2`
- Do not worry about it ... right now.

Coding conventions

Documentation strings (I)

Documentation is important:

- Q: Will you remember why did you wrote that crazy code line?
- A: No, so you must document your code.
- A: Yes, no programmer likes documentating his code.

Python provides automatic documentation features:

- It can be accessed with `foo.__doc__` (version 3.X)

```
>>> print(print.__doc__)
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Prints the values to a stream, or to `sys.stdout` by default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the current `sys.stdout`.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

Coding conventions

Documentation strings (II)

Documentation conventions:

- The first line should be a summary.
- The second line should be blank.
- One or more lines with detailed description (arguments, side effects, etc).
- Respect indentation.

```
def my_function():
    """Do nothing, but document it.

    No, really, it doesn't do anything
    """
    pass

print(my_function.__doc__)
```

Coding conventions

Coding style (I)

Make your code easy to read using good coding style.

Python coding style convention:

- 4-space indentation, with no tabs.
- Maximum 79 characters per code line.
- Separate functions and classes with white lines.
- Separate large code blocks with white lines.
- Use docstrings.
- Operators spacing: `a = f(1, 2) + g(3, 4)`.
- Proper use of capitals:
 - Classes: `CamelCase`
 - Methods and functions: `lower_case_with_underscores()`

Want to know more? [Click here!](#)

Examples

Example 1: Matrices addition

```
X = [[12, 7, 3],
      [4, 5, 6],
      [7, 8, 9]]

Y = [[5, 8, 1],
      [6, 7, 3],
      [4, 5, 9]]

result = [[0, 0, 0],
          [0, 0, 0],
          [0, 0, 0]]

# iterate through rows
for i in range(len(X)):
    # iterate through columns
    for j in range(len(X[0])):
        result[i][j] = X[i][j] + Y[i][j]

for r in result:
    print(r)
```


Examples

Example 2: Calculator

```
def add(x, y):  
    """This function adds two numbers"""  
    return x + y  
  
def subtract(x, y):  
    """This function subtracts two numbers"""  
    return x - y  
  
def multiply(x, y):  
    """This function multiplies two numbers"""  
    return x * y  
  
# take input from the user  
print("Select operation.")  
print("1. Add")  
print("2. Subtract")  
print("3. Multiply")  
  
choice = input("Enter choice (1/2/3):")  
num1 = int(input("Enter first number: "))  
num2 = int(input("Enter second number: "))  
  
if choice == '1':  
    print(num1, "+", num2, "=", add(num1, num2))  
elif choice == '2':  
    print(num1, "-", num2, "=", subtract(num1, num2))  
elif choice == '3':  
    print(num1, "*", num2, "=", multiply(num1, num2))  
else:  
    print("Invalid input")
```

Bibliographic references I



[van Rosum, 2012] G. van Rossum, Jr. Fred L. Drake.
Python Tutorial Release 3.2.3, chapter 4.
Python Software Foundation, 2012.



[Bahit, 2008] E. Bahit.
Curso: Python para principiantes.
Creative Commons Atribución-NoComercial 3.0, 2012.



[Swaroop, 2008] C H. Swaroop.
A Byte of Python.
Creative Commons Attribution-ShareAlike 3.0, 2008.



[Pilgrim, 2004] M. Pilgrim.
Dive into Python.
Ed. Prentice Hall, 2004.