9_funzioni

June 1, 2023

[ ]:
```
# Functions.

# Partitioning (or "factoring") the code into smaller units is useful to:
# - reuse a portion of code several times in the same program or in different
#   programs;
# - reduce unwanted "interactions" between different parts of codes (see e.g.
#   variable's scope);
# - make the code easier to read, debug, and maintain;
# - for large codes, work on the same project as a team.

# Definition.

def f(x):  # x is a dummy argument
    r = x * x
    return r

# Calling, passing arguments.

a = 2.0
b = f(a)  # a is the actual argument
print(b)
print(f(a))  # function composition works

# Most important: when the function is defined its code is not executed.
# Also: the name of the _actual_ argument to the function at execution time
# need not be the same as the name of the _dummy_ argument in the function's
# definition.

# Multiple arguments.

def f(x,y):
    r = x * y
    return r

print(f(2.0,3.0))

# x and r belong to the _local scope_ of the function only.
```

```python
#print(r)  # error


# Variables' scope.
# The "scope" of a variable is the portion of the code where the variable is
# available.

# Use a variable in the global scope.

g = 2.0

def f(x):
    r = g * x
    return r

print(f(2.0))

g = 3.0

print(f(2.0))

# Assign a variable in the global scope.  (Avoid.)

g = 2.0

def f(x):
    global g
    r = g * x
    g = g / 2.0
    return r


print(f(2.0))
print(f(2.0))  # the same function gives a different result
print(g)  # the global variable has been modified implicitly by f



# Variables scope and mutable types.

l = [1, 2, 3]

def act_on_list_wrong(u,x):
    u = u + [x]  # the assignment makes u a variable in the local scope
    return u
```

```
l1 = act_on_list_wrong(l, 4)
print(l)  # the global list is not modified
print(l1)

def act_on_list_right(u,x):
    u.append(x)
    return u

l1 = act_on_list_right(l, 4)
print(l)  # the global list is modified
print(l1)

# Note: this will be clearer when we discuss classes and objects.


# Passing arguments.

# Multiple arguments.

def rectangle(base, height):
    area = base * height
    return area

print(rectangle(2.0, 3.0))

# Optional, keywords arguments, default values.

def rectangle(base=1.0, height=7.0):
    area = base * height
    return area

print(rectangle(2.0, 3.0))

print(rectangle())

print(rectangle(height=3.0))

print(rectangle(base=2.0, height=3.0))

print(rectangle(height=3.0, base=2.0))

print(rectangle(2.0, height=3.0))

d = {"base": 2.0, "height": 3.0}
print(rectangle(**d))  # dictionary _unpacking_

d = {"height": 3.0}
```

```
print(rectangle(base=2.0, **d))


# Example.
# Ask the user for measurements output until "end" is entered.
# Print the average.

def get_new_number(nn):
    s = input("Please enter a number: ")
    if (s == "end"):
        r = False
    else:
        xx.append(float(s))  # one should make sure that it is a float
        r = True
    return r


xx = []
keep_asking = True
while keep_asking:
    keep_asking = get_new_number(xx)


if (len(xx) > 0):
    av = 0.0
    for x in xx:
        av = av + x
    av = av / len(xx)
    print("The average is: %8.3f" % (av))


# Pass functions as arguments.
# Functions are "first-class citizens" in Python. In particular, they can be
# passed as arguments to other functions.

# Calculate the definite integral of a generic function g.
def integral(g, x0, x1, n):
    dx = (x1 - x0) / n
    xx = [(x0 + i * dx) for i in range(n)]
    # Use the function g.
    yy = [g(x) for x in xx]
    s = 0.0
    for y in yy:
        s = s + y
    s = s * dx
    # Note: numerically it is less time-consuming to multiply by dx just once!
    return s
```

```
# Define an integrand.
def f(x):
    r = x * x
    return r

# Define the boundaries of the integration domain.
xL = 0.0
xR = 1.0

print("The numerical result is %8.5f" % (integral(f, xL, xR, 200)))
print("The analytical result is %8.5f" % ((xR**3.0 - xL**3.0)/3.0))


# Recursive functions.  (Use with caution!)

def factorial(n):
    # This is where the "forking" of the function stops.
    if (n == 1):
        r = 1
    else:
        # We can call the same function that we are defining!
        r = n * factorial(n - 1)
    return r

print(factorial(3))
print(factorial(4))


# Anonymous functions.

# Define a function with the notation "lambda".
f = lambda x: x*x

def f(x):
    return x*x # equivalent

print(f(3))

f = lambda x, y: x*y
print(f(3.0, 2.0))


# Use as argument to another function.
def g(f):
    r = f(2.0)
    return r
```

```python
print(g(lambda y: y*y)) # equivalent to g(f), f(y) = y*y


# Assign to list elements.  (Functions are just variables!)
powers = [
    lambda x: 1.0,
    lambda x: x,
    lambda x: x**2.0,
    lambda x: x**3.0
]

x = 2.0
for f in powers:
    print(f(x))


# Note that the parameters are evaluated at execution time.
# Problem of the "closure":
# https://en.wikipedia.org/wiki/Closure_(computer_programming)
powers_wrong = []
for n in range(4):
    powers_wrong.append(lambda x: x**n)
# All the anonymous functions are still "linked" to the variable n, so its
# actual value is not saved.

x = 2.0
for f in powers_wrong:
    print(f(x))  # it always print 2^(current value of n)

# Solve this problem by having the parameter as the dummy variable of another
# function, so that the parameter is not in the scope when the anonymous
# function is executed.
def create_power(n):
    f = lambda x: x**n
    return f
# When f is returned, the "link" to n is lost and the actual value of n
# is saved.

powers = []
for n in range(4):
    powers.append(create_power(n))

x = 2.0
for f in powers:
    print(f(x))  # it prints 2^(value of n when create_power was executed)
```

```
4.0
```

```
4.0
6.0
4.0
6.0
4.0
2.0
0.5
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3, 4]
[1, 2, 3, 4]
6.0
6.0
7.0
3.0
6.0
6.0
6.0
6.0
6.0
The average is:    2.000
The numerical result is  0.33084
The analytical result is  0.33333
6
24
9
6.0
4.0
1.0
2.0
4.0
8.0
8.0
8.0
8.0
8.0
1.0
2.0
4.0
8.0
```

RIASSUNTO:

FUNZIONI 1. vengono definite in funzione di una o più variabili con le operazioni "def" e "return" 2. scope di una variabile: porzione di codice in cui la variabile è visibile: è importante evitare di utilizzare il global scope di una variabile all'interno di una funzione perché poi ogni volta che tale funzione viene chiamata, e se al suo interno esiste un comando per la variabile, la variabile stessa viene modificata 3. il modo giusto di modificare una lista tramite funzioni non è definendo una funzione che fa la somma (concatenazione) di liste, bensì è il metodo append() 4. è possibile calcolare l'area di un oggetto, sia passando solo le variabili, sia passando anche dei valori di default, o anche tramite il "dictionary unpacking" (**d) usando come parametri della funzione degli elementi di un dizionario 5. possono essere utilizzate per fare calcoli complicati, come gli integrali o i fattoriali 6. funzioni anonime (lambda): la comodità è che possono essere definite in una sola riga e godono comunque delle proprietà delle funzioni normalmente definite, come la possibilità di essere chiamate. Tuttavia, l'effetto collaterale è che, se le funzioni lambda utilizzano una variabile definita tramite un ciclo, la funzione assumerà sempre lo stesso valore per l'ultimo valore del ciclo, in quanto la funzione lambda è legata alla variabile e non al suo valore