
# 15-110 Principles of Computing – S19

## Lab 11

Teacher:
Gianni A. Di Caro

# How do we pass arguments to a function?

```
def function_name(arguments):
    function_body
    return something
```

- If we only need to pass one **single argument**, there's no much to say
- The situation is different when passing **multiple parameters** …

```
def quadratic_roots(a, b, c):
    x1 = -b / (2 * a)
    x2 = sqrt(b**2 - (4 * a *c)) / (2 * a)
    return (x1 + x2), (x1 - x2)
```

✓ Passing arguments as **positional arguments**

```
quadratic_roots(31, 93, 62)
```

is different than:

```
quadratic_roots(93, 31, 62)
```

**Order matters!**

✓ Passing arguments as **keyword arguments**

```
quadratic_roots(a=31, b=93, c=62)
```

is the same as:

```
quadratic_roots(b=93, a=31, c=62)
```

**Order doesn't matter!**

# Keyword arguments and the `help()` function

- Passing arguments as **keyword arguments** works because python **knows the name function arguments**, and therefore it can perform automatic matching without errors

```
quadratic_roots(a=31, b=93, c=62)
quadratic_roots(b=93, a=31, c=62)
quadratic_roots(c=62, b=93, a=31)
```

all give the same result, (-1.0, -2.0) in this case

→ We can ask python **help** on function's parameters using the `help(function_name)` function:

`help(quadratic_roots)` would give as answer: `quadratic_roots(a,b,c)`

Use of keyword arguments increases the clarity of a program!

`random_password(upper=1, lower=1, digits=1, length=8)` vs. `random_password(1, 1, 1, 8)`

# Positional arguments: different possible errors

- When passing arguments as **positional arguments** we need to be careful to **match the order** with which the parameters appear in the function definition!

  - Wrong computations (no errors are issued by the interpreter!)

```
quadratic_roots(31, 93, 62) → (-1.0, -2.0)

quadratic_roots(62, 93, 31) → (-0.5, -1.0)
```

  - Run-time errors due to incorrect type (program aborted!)

```
def sing(person, repetitions):
    for i in range(repetitions):
        happy()
        happy()
        print("Happy Birthday, dear", person + ".")
        happy()
```

```
sing(2, “Fred”)
```

Throws an error because a string object cannot be interpreted as an integer

# Default values for the arguments, equivalent function calls

- When defining a function, a **default value can be defined for each argument**
  - If a value argument for an argument with a default value is passed (either by position or by keyword) when the function is called, then the argument takes the provide value
  - Otherwise, the argument takes the default value

- `def sing(person="Fred", repetitions=2):`
  - ✓ `sing()`
  - ✓ `sing("Lucy")`
  - ✓ `sing("Lucy", 3)`
  - → NO: `sing(3)`

- `def sing(person, repetitions=2):`
  - → NO: `sing()`
  - ✓ `sing("Lucy")`
  - ✓ `sing("Lucy", 3)`
  - → NO: `sing(3)`

→ NO: `def sing(person="Fred", repetitions):` parameter assignments would be ambiguous

# Default values for making  arguments optional

- The parameters with default values are de facto **optional**, in the absence of them they take the default value, that might be an empty value

  - `def sing(person, repetitions = 2):`
  - `def draw_rectangle(x1, x2, y1, y2, fill_color = None):`
  - `def move_forward(distance, velocity = 10):`

# Arbitrary number of arguments

- It might be the case that a function does some repetitive job and operates on a non well-defined number of arguments
- E.g., `print()` function
- We could use lists but it's not always nice, convenient, appropriate pack everything into a string
- Arbitrary sequence of arguments can be passed with the notation `*arguments`

```
def longlen(*strings):
    max = 0
    for s in strings:
      if len(s) > max:
         max = len(s)
    return max

longlen('apple','banana','cantaloupe','cherry')  → 9
longlen('red','blue','green')                    → 5
```

- Positional and keyword arguments should be placed first to the arbitrary ones

```
def my_func(a, b=True, *args):
```

# Equivalent function calls by mixing positional and keyword arguments

- When passing arguments as **positional arguments** we need to be careful to **match the order** with which the parameters appear in the function definition!

  - Wrong computations (no errors are issued by the interpreter!)

```
quadratic_roots(31, 93, 62) → (-1.0, -2.0)

quadratic_roots(62, 93, 31) → (-0.5, -1.0)
```

  - Run-time errors due to incorrect type (program aborted!)

```
def sing(person, repetitions):
    for i in range(repetitions):
        happy()
        happy()
        print("Happy Birthday, dear", person + ".")
        happy()
```

```
sing(2, “Fred”)
```

Throws an error because a string object cannot be interpreted as an integer

# Passing functions as arguments!

- Function arguments can also include a **function, that can be then regularly invoked inside the function**

```
def parabola(x):
    return -x*x + 2

def cubic(x):
    return x*x*x + 2*x*x

def geometric(x):
    return 1 / (1-x)

def line(x):
    return x
```

```
def estimate_max_in_interval(f, low_val, high_val, samples):
    x = low_val
    step = (high_val - low_val) / samples
    max_val = f(high_val)
    for i in range(samples):
        if f(x) > max_val:
            max_val = f(x)
        x += step
    return max_val

print(estimate_max_in_interval(geometric, -2, -1, 100))
```