

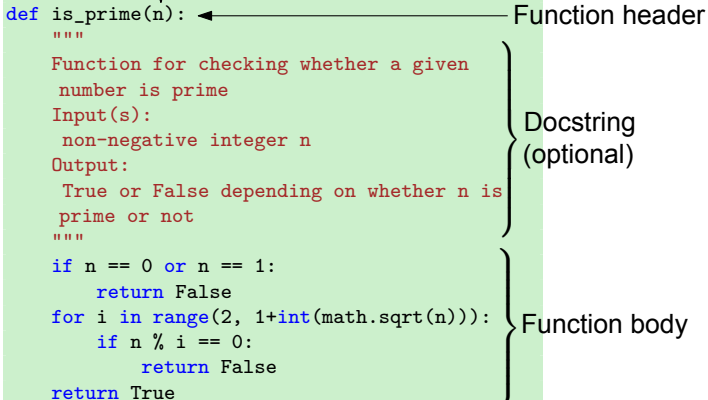
# Python: functions

Intro. to computing

`https://mandarm.github.io/computing.html`

# Functions I

Function parameters (zero or more, comma separated)



```
def is_prime(n):  
    """  
    Function for checking whether a given  
    number is prime  
    Input(s):  
        non-negative integer n  
    Output:  
        True or False depending on whether n is  
        prime or not  
    """  
    if n == 0 or n == 1:  
        return False  
    for i in range(2, 1+int(math.sqrt(n))):  
        if n % i == 0:  
            return False  
    return True
```

The diagram illustrates the components of a Python function definition. A green box highlights the code. Annotations include: an arrow pointing to the parameter `n` with the text "Function parameters (zero or more, comma separated)"; a line pointing to the `def` keyword and parameter list with the label "Function header"; a bracket on the right side of the docstring with the label "Docstring (optional)"; and a bracket on the right side of the function body with the label "Function body".

# Functions II

- *Parameters* (or formal parameters): `n` in the above example
- *Arguments* (or actual parameters): value passed when the function is actually called

Example: `is_prime(4)`, `is_prime(5)` — 4 and 5 are arguments

- *return* statements
  - may be followed by an optional expression (including `None`)
  - multiple occurrences permitted (as in example above)
  - no `return` statement OR no expression after `return`  
⇒ function returns `None`
- *Function call signature*: number and type of arguments + return value  
(cf.  $f : D_1 \times D_2 \times \dots \times D_k \rightarrow R$ )

# Naming conventions

- Use lowercase letters
- Use underscore (\_) as word separator

**Example:** `read_data()` preferred to `readData()`

# Parameters / arguments

- Number of arguments: fixed vs. variable
- Type of arguments: positional vs. named
- Type of arguments: required (compulsory) vs. optional
- **Special markers:** `/`, `*`, `**`

# Variadic parameters

```
def multiply2(x, y): # 2 arguments
    return x * y

def multiply(*args): # could also use *numbers
    product = 1
    for x in args:
        product *= x
    return product

def multiply_flexible(initial, *numbers):
    product = initial
    for x in numbers:
        product *= x
    return product
```

- **Variadic parameter**  $\equiv$  accepts variable number of arguments
- Parameter name prefixed with \*, e.g., `*args`, `*numbers`
- Parameter becomes a tuple containing *remaining* (0 or more) arguments

# Variadic parameters

```
def multiply2(x, y): # 2 arguments
    return x * y

def multiply(*args): # could also use *numbers
    product = 1
    for x in args:
        product *= x
    return product

def multiply_flexible(initial, *numbers):
    product = initial
    for x in numbers:
        product *= x
    return product
```

What happens if you swap  
initial and \*numbers

- **Variadic parameter**  $\equiv$  accepts variable number of arguments
- Parameter name prefixed with \*, e.g., `*args`, `*numbers`
- Parameter becomes a tuple containing *remaining* (0 or more) arguments

# Positional vs. named arguments I

```
def multiply_flexible(*numbers, initial):  
    product = initial  
    for x in numbers:  
        product *= x  
    return product  
  
>>> multiply_flexible(2,3,4,10)           # carefully study error message  
>>> multiply_flexible(2,3,4,initial=10)    # 240
```

- **All** arguments preceding *non-empty* `*args` must be passed by position.
- **All** arguments following `*args` must be named arguments, i.e., passed using `parameter_name=value` format.

# Positional vs. named arguments II

```
def example(x, y, *, a, b, c):  
    print("x =", x, "\ny =", y, "\na =", a, "\nb =", b, "\nc =", c)  
  
example(1,2,3) # ERROR: takes 2 positional arguments but 3 were given  
example(1,2,b=3,c=4,a=5)      # OK  
example(y=2,x=1,b=3,c=4,a=5)  # OK  
example(b=3,c=4,a=5,y=2,x=1)  # OK
```

order of named arguments does not matter

- \* (without a parameter name) does not match anything.
- Arguments preceding \* may be passed by position or as named argument, i.e., using `parameter_name=value` format.
- All arguments following *first* named argument must also be named arguments, i.e., all positional arguments must precede all named arguments.
- All arguments following \* must be named arguments.

This rule applies even when a variadic parameter (\* or \*args) is not used.

# Positional vs. named arguments III

## Positional-only arguments

- Can use `/` to force *preceding* arguments to be passed as positional arguments only  
(introduced in Python 3.8)

### Digression: `help()` function

Run `help(variable or function name)` to see inbuilt documentation on the corresponding type or function.

### Examples:

```
>>> a = [2,1]
>>> help(a)
>>> help(len)    # len(obj, /)
>>> help(sum)    # sum(iterable, /, start=0)
```

# Summary: variadic, positional, and named parameters

Positional-only if at least one argument is provided for args.  
Positional / named if no argument is provided for args.

Positional only  
def function\_name(a, b, c, ..., /, p, q, r, ..., \*args, v = val1, w = val2, ..., \*\*kwargs)

Named only

Optional; anything to the left of this is **positional** only; **cannot** be passed in name=value form.

- Optional
- kwargs  $\equiv$  dict containing all remaining arguments (must be in name=value form)
- Optional: anything to the right of this is **named** only; cannot be passed by position
- args  $\equiv$  tuple containing all arguments upto and excluding first name=value type argument
- args omitted  $\Rightarrow$  passing arguments is an error
- if one or more argument(s) is supplied for args, all preceding arguments must be **positional**

- *When calling*, all positional arguments must precede all named arguments.

# Required vs. optional arguments

- **Optional parameter**  $\equiv$  parameter for which default value is provided in function header

When calling the function, caller can

- omit the argument
- provide a value (same as / different from the default)
- *All parameters following first optional parameter must also be optional, with defaults provided.*
- Defaults **cannot** be provided for variadic (or preceding) parameters.
- Default parameter values evaluated **once** when function is defined
  - using mutable objects as default may lead to unexpected results

```
def func(x, items=[]): # 2nd call onwards, items is non-empty
    items.append(x)
    return items
```

$\Rightarrow$  preferable to use only immutable objects as default values

# More examples

```
def example2(x, y, *gobble, a=10, b=20, c=30):  
    print("x =", x, "\ny =", y, "\na =", a, "\nb =", b, "\nc =", c)  
    print("gobble =", *gobble)
```

```
example2(x=1,y=2)
```

```
x = 1  
y = 2  
a = 10  
b = 20  
c = 30  
gobble =
```

```
example2(1,2,3,4,5)
```

```
x = 1  
y = 2  
a = 10  
b = 20  
c = 30  
gobble = 3 4 5
```

# Digression: \* and \*\* as unpacking operators

- \*: *unpack* a sequence
- \*\*: *unpack* a dictionary into tuples

```
def example3(a, b, c, param1, param2, param3, param4):  
    print("a =", a,  
          "b =", b,  
          "c =", c,  
          "param1 =", param1,  
          "param2 =", param2,  
          "param3 =", param3,  
          "param4 =", param4)
```

```
>>> t = [ 1, 2, 3 ]  
>>> d = {"param1" : 'a', "param2" : 'b', "param3" : 'c', "param4" : 'd'}  
>>> example(t, d)          # ERROR  
>>> example(*t, **d)       # works as expected
```

# Things to keep in mind

- All required arguments must be provided (cannot be omitted).
- Multiple values should not be provided for any parameter.
- `name` part of any `name = value` argument must match a parameter name.
- Arguments are fully evaluated left-to-right before executing function body.

## ***Parameter passing, return values***

# Parameter passing

- Function parameters  $\equiv$  names that are *references* to the actual input objects
  - $\Rightarrow$  function works on 'original' input objects, **not copies**
  - $\Rightarrow$  changes made to mutable arguments are visible in the calling function

```
def reverse_in_place(L):  
    for i in range(len(L) // 2):  
        L[i], L[-i-1] = L[-i-1], L[i]  
  
def sum_squares(L):  
    L = [ x*x for x in L ]      # NOTE: local L is reassigned to a NEW  
                                list  
    return sum(L)  
  
>>> L = [1,2,3]  
>>> reverse_in_place(L)  
>>> sum_squares(L)           # L is not changed  
>>> L
```

# Return values

- To return multiple values, place them in a tuple.
- Values returned in a tuple can be unpacked to individual variables.

```
def quotient_and_remainder(dividend, divisor):  
    return (dividend//divisor, dividend % divisor) # brackets optional  
  
q, r = quotient_and_remainder(25, 7)
```

## ***Scope rules***

# Local names

- All function parameters
  - *All variables that are assigned in function body* (see example below)
  - All other names (used but **not** assigned in function body)  $\equiv$  global names (must exist)
  - `global x`  $\equiv$  `x` refers to global name, *even if `x` is assigned* in function body
- generally considered to be bad style / avoidable

```
x = 10
def example5():
    print(x)      # x not assigned => global
>>> example5()   # prints 10

def example5():
    print(x)      # x not bound yet
    x = 20        # x is assigned => local
>>> example5()   # ERROR: UnboundLocalError
```

```
def example6():
    global x
    print(x)
    x = 20
>>> example6()
>>> x      # updated to 20
```

Will not cover: nested functions,  
`nonlocal`

## ***Anonymous / lambda functions***

# Lambda functions I

- Anonymous function
- Takes any number of comma-separated arguments
- Body must consist of a *single* expression ( $\equiv$  return value)  
(multiple statements, non-expression statements not allowed)

## Examples:

```
import math
# Names given to anonymous functions for convenience
x = lambda theta: math.sin(theta) + math.cos(theta)
y = lambda theta: math.sin(theta + math.pi/4) * (2 ** 0.5)
print(x(math.pi/3), y(math.pi/3))

l1 = [ (1893, "Mahalanobis"), (1890, "Fisher"), (1777, "Gauss"), \
       (1894, "Neyman"), (1920, "CR Rao") ]
l2 = [ ("Mahalanobis", 1893), ("Gauss", 1777), ("Fisher", 1890), \
       ("Neyman", 1894), ("CR Rao", 1920) ]
sorted(l1)
sorted(l2)
```

# Lambda functions II

```
sorted(l1, key=lambda x: x[1])  
sorted(l1, key=lambda x: len(set(x[1])))  
sorted(l2, key=lambda x: x[1])
```

```
def rh_derivative(f, x):  
    h = 1e-8  
    return (f(x+h) - f(x))/h  
  
rh_derivative(math.sqrt, 4)  
rh_derivative(lambda x: 3*x*x + 2*x, 10)
```

# Globals in lambdas

- Value used  $\equiv$  current value when lambda is *called* (**not** value at definition time)
- To use value at the time of definition, use it as default value

```
x = 2
f = lambda y, x=x: x * y
x = 3
g = lambda y, x=x: x * y
print(f(10))      # 20
print(g(10))      # 30
```

# References

- *Python in a Nutshell*, 4th ed., Chapter 3, last section (Functions).
- *Python Distilled*, Chapter 5.