PROGRAMMING IN PYTHON I

Functions

Andreas Schörgenhumer
Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

Institute for
Machine Learning

Copyright Statement

Contact

Andreas Schörgenhumer

Institute for Machine Learning
Johannes Kepler University
Altenberger Str. 69
A-4040 Linz

E-Mail: `schoergenhumer@ml.jku.at`

Write mails only for personal questions

Institute ML Homepage

FUNCTIONS

Motivation

- Often, we encounter similar problems multiple times

Motivation

- Often, we encounter similar problems multiple times
- We need to perform the same sequence of operations repeatedly but possibly with different input

Motivation

- Often, we encounter similar problems multiple times
- We need to perform the same sequence of operations repeatedly but possibly with different input
- However, we do not want to repeat/copy code! Why are redundancies bad (code duplication)?

Motivation

- Often, we encounter similar problems multiple times
- We need to perform the same sequence of operations repeatedly but possibly with different input
- However, we do not want to repeat/copy code! Why are redundancies bad (code duplication)?
  - □ Prone to errors

Motivation

- Often, we encounter similar problems multiple times
- We need to perform the same sequence of operations repeatedly but possibly with different input
- However, we do not want to repeat/copy code! Why are redundancies bad (code duplication)?
  - Prone to errors
  - Make program long, which means more to read

Motivation

- Often, we encounter similar problems multiple times
- We need to perform the same sequence of operations repeatedly but possibly with different input
- However, we do not want to repeat/copy code! Why are redundancies bad (code duplication)?
  - Prone to errors
  - Make program long, which means more to read
  - More difficult to maintain (need to change all relevant code parts for updates)

Example

- Find maximum of x and y and store it in a result variable

```
x = 4
y = 5
maximum = x
if y > maximum:
    maximum = y

... # different values assigned to x, y

maximum = x
if y > maximum:
    maximum = y
```

Example

- Find maximum of x and y and store it in a result variable

```
x = 4
y = 5
maximum = x
if y > maximum:
    maximum = y

... # different values assigned to x, y

maximum = x
if y > maximum:
    maximum = y
```

- Extract common code and make input parameterizable → **functions**

Solution with Function

- Preferred solution: function with input and output
  - □ Parameters: take the two values as input
  - □ Output: return the maximum

```
def get_max(x, y):
    maximum = x
    if y > maximum:
        maximum = y
    return maximum
```

Solution with Function

- Preferred solution: function with input and output
    - Parameters: take the two values as input
    - Output: return the maximum

```python
def get_max(x, y):
    maximum = x
    if y > maximum:
        maximum = y
    return maximum
```

- Can use this function now multiple times:

```python
max1 = get_max(4, 5)
max2 = get_max(9, 0)
...
```

Solution with Function

- Preferred solution: function with input and output
  - Parameters: take the two values as input
  - Output: return the maximum

```
def get_max(x, y):
    maximum = x
    if y > maximum:
        maximum = y
    return maximum
```

- Can use this function now multiple times:

```
max1 = get_max(4, 5)
max2 = get_max(9, 0)
...
```

- In case of changes or error fixing, only one code part to check (much better maintainability)

Functions in Python

- Functions can have input via **parameters** and they can **return** values (both are optional)
- Naming convention is equal to variables (lowercase letters + underscores if needed)

```python
def fun1():
    # do something (side effects)

def fun2():
    # create some result and return value
    return result

def fun3(x):
    # do something with x (side effects)

def fun4(x, y, z):
    # do something with x, y, z and return value
    return (x + y) * z
```

Formal and Actual Parameters

- **Formal parameters** are those specified in the function definition
- **Actual parameters** (often called **arguments**) are those specified when calling the function

Formal and Actual Parameters

- **Formal parameters** are those specified in the function definition
- **Actual parameters** (often called **arguments**) are those specified when calling the function
- Example:

```
def get_max(x, y):
    return x if x > y else y

max1 = get_max(4, 5)
max2 = get_max(-12, my_var)
```

- `x` and `y` are formal parameters, `4` and `5`, and `-12` and `my_var` are actual parameters (arguments)

Argument Passing

- The actual parameters are passed to the function using **call by value** with value = **object reference**

Argument Passing

- The actual parameters are passed to the function using **call by value** with value = **object reference**
  - Analogous to variable assignment

Argument Passing

- The actual parameters are passed to the function using **call by value** with value = **object reference**
  - Analogous to variable assignment
  - E.g., `a = 2` and `b = a` → both refer to the same object

Argument Passing

- The actual parameters are passed to the function using **call by value** with value = **object reference**
  - Analogous to variable assignment
  - E.g., `a = 2` and `b = a` → both refer to the same object
- Important when dealing with **mutable** objects (e.g., lists) since changes done within the function will be reflected outside!

Argument Passing

- The actual parameters are passed to the function using **call by value** with value = **object reference**
  - □ Analogous to variable assignment
  - □ E.g., `a = 2` and `b = a` → both refer to the same object
- Important when dealing with **mutable** objects (e.g., lists) since changes done within the function will be reflected outside! Example:

```python
def add_to_list(some_list, item):
    # append directly changes the list in-place
    some_list.append(item)

my_list = [1, 2, 3]
print(my_list)  # [1, 2, 3]
add_to_list(my_list, 4)  # some_list = my_list
print(my_list)  # [1, 2, 3, 4]
```

Positional and Keyword Arguments

- Arguments can be passed as **positional arguments** or **keyword arguments**

Positional and Keyword Arguments

- Arguments can be passed as **positional arguments** or **keyword arguments**
- Keyword arguments can be at any position and in an arbitrary order

Positional and Keyword Arguments

- Arguments can be passed as **positional arguments** or **keyword arguments**
- Keyword arguments can be at any position and in an arbitrary order
- Positional arguments must not appear after a keyword argument

Positional and Keyword Arguments

- Arguments can be passed as **positional arguments** or **keyword arguments**
- Keyword arguments can be at any position and in an arbitrary order
- Positional arguments must not appear after a keyword argument
- Example:

```
def get_max(x, y):
    return x if x > y else y
```

Positional and Keyword Arguments

- Arguments can be passed as **positional arguments** or **keyword arguments**
- Keyword arguments can be at any position and in an arbitrary order
- Positional arguments must not appear after a keyword argument
- Example:

```
def get_max(x, y):
    return x if x > y else y
```

  - □ `get_max(4, 5)` → x, y positional (x=4, y=5)

Positional and Keyword Arguments

- Arguments can be passed as **positional arguments** or **keyword arguments**
- Keyword arguments can be at any position and in an arbitrary order
- Positional arguments must not appear after a keyword argument
- Example:

```
def get_max(x, y):
    return x if x > y else y
```

  - □ `get_max(4, 5)` → x, y positional (x=4, y=5)
  - □ `get_max(4, y=5)` → x positional (x=4), y keyword

Positional and Keyword Arguments

- Arguments can be passed as **positional arguments** or **keyword arguments**
- Keyword arguments can be at any position and in an arbitrary order
- Positional arguments must not appear after a keyword argument
- Example:

```python
def get_max(x, y):
    return x if x > y else y
```

  - □ `get_max(4, 5)` → x, y positional (x=4, y=5)
  - □ `get_max(4, y=5)` → x positional (x=4), y keyword
  - □ `get_max(x=4, y=5)` → x, y keyword

Positional and Keyword Arguments

- Arguments can be passed as **positional arguments** or **keyword arguments**
- Keyword arguments can be at any position and in an arbitrary order
- Positional arguments must not appear after a keyword argument
- Example:

  ```python
  def get_max(x, y):
      return x if x > y else y
  ```

  - `get_max(4, 5)` → x, y positional (x=4, y=5)
  - `get_max(4, y=5)` → x positional (x=4), y keyword
  - `get_max(x=4, y=5)` → x, y keyword
  - `get_max(y=4, x=5)` → x, y keyword (different order)

Positional and Keyword Arguments

- Arguments can be passed as **positional arguments** or **keyword arguments**
- Keyword arguments can be at any position and in an arbitrary order
- Positional arguments must not appear after a keyword argument
- Example:

  ```python
  def get_max(x, y):
      return x if x > y else y
  ```

  - `get_max(4, 5)` → x, y positional (x=4, y=5)
  - `get_max(4, y=5)` → x positional (x=4), y keyword
  - `get_max(x=4, y=5)` → x, y keyword
  - `get_max(y=4, x=5)` → x, y keyword (different order)
  - `get_max(y=4, 5)` → not allowed (pos arg after kw arg)

Variable Arguments (1)

- You can specify your function to allow arbitrary many arguments (also zero):[1]
  - □ For positional arguments: ***args**. Every argument will be collected in a tuple.
  - □ For keyword arguments: ****kwargs**. Every argument will be collected in a dictionary.

[1] `args` and `kwargs` are typically used names

Variable Arguments (1)

- You can specify your function to allow arbitrary many arguments (also zero):[1]
  - For positional arguments: ***args**. Every argument will be collected in a tuple.
  - For keyword arguments: ****kwargs**. Every argument will be collected in a dictionary.

```
def fun(*args, **kwargs):
    # Do something with tuple args
    # Do something with dict kwargs
```

[1] `args` and `kwargs` are typically used names

Variable Arguments (1)

- You can specify your function to allow arbitrary many arguments (also zero):[1]
  - □ For positional arguments: ***args**. Every argument will be collected in a tuple.
  - □ For keyword arguments: ****kwargs**. Every argument will be collected in a dictionary.

```
def fun(*args, **kwargs):
    # Do something with tuple args
    # Do something with dict kwargs
```

- Example call with `fun(1, 2, x=3, y=4)`:

```
args = (1, 2)
kwargs = {"x": 3, "y": 4}
```

[1] `args` and `kwargs` are typically used names

Variable Arguments (2)

- Variable number of arguments can be mixed with normal parameter specification under the following rules:

Variable Arguments (2)

- Variable number of arguments can be mixed with normal parameter specification under the following rules:
  - Zero or more normal parameters can occur before `*args`

Variable Arguments (2)

- Variable number of arguments can be mixed with normal parameter specification under the following rules:
  - Zero or more normal parameters can occur before `*args`
  - Zero or more normal parameters can occur after `*args` but they will only be accessible via “keyword-only arguments”

Variable Arguments (2)

- Variable number of arguments can be mixed with normal parameter specification under the following rules:
  - Zero or more normal parameters can occur before `*args`
  - Zero or more normal parameters can occur after `*args` but they will only be accessible via “keyword-only arguments”
  - No parameters can follow `**kwargs`

Variable Arguments (2)

- Variable number of arguments can be mixed with normal parameter specification under the following rules:
  - Zero or more normal parameters can occur before `*args`
  - Zero or more normal parameters can occur after `*args` but they will only be accessible via “keyword-only arguments”
  - No parameters can follow `**kwargs`
  - Normal parameters are never part of `*args` or `**kwargs`

Variable Arguments (2)

- Variable number of arguments can be mixed with normal parameter specification under the following rules:
  - □ Zero or more normal parameters can occur before `*args`
  - □ Zero or more normal parameters can occur after `*args` but they will only be accessible via “keyword-only arguments”
  - □ No parameters can follow `**kwargs`
  - □ Normal parameters are never part of `*args` or `**kwargs`
- Example with a few function calls afterwards:

```
def fun(x, *args, y, **kwargs):
    # Do something

fun(1)              # Error: missing kw-only arg y
fun(1, 2)           # Error: missing kw-only arg y
fun(1, y=2)         # x=1, args=(), y=2, kwargs={}
fun(1, 3, y=2)      # x=1, args=(3,), y=2, kwargs={}
fun(1, z=4, y=2) # x=1, args=(), y=2, kwargs={"z":4}
```

Parameter Unpacking

- You can **unpack** values when calling functions:

Parameter Unpacking

- You can **unpack** values when calling functions:
  - Use the ∗ operator to unpack iterable (e.g., list) elements that are then passed to the function as positional arguments

Parameter Unpacking

- You can **unpack** values when calling functions:
    - Use the * operator to unpack iterable (e.g., list) elements that are then passed to the function as positional arguments
    - Use the ** operator to unpack dictionary elements that are then passed to the function as keyword arguments

Parameter Unpacking

- You can **unpack** values when calling functions:
  - Use the `*` operator to unpack iterable (e.g., list) elements that are then passed to the function as positional arguments
  - Use the `**` operator to unpack dictionary elements that are then passed to the function as keyword arguments
- Can mix normal argument specification and unpacking (restriction: `*` must not occur after `**`)

Parameter Unpacking

- You can **unpack** values when calling functions:
  - □ Use the `*` operator to unpack iterable (e.g., list) elements that are then passed to the function as positional arguments
  - □ Use the `**` operator to unpack dictionary elements that are then passed to the function as keyword arguments
- Can mix normal argument specification and unpacking (restriction: `*` must not occur after `**`)
- Example:

```
def fun(x, *args, y, **kwargs):
    # Do something

my_list = [1, 2, 3]
my_dict = {"y": 4, "z": 5}
fun(*my_list, **my_dict)
# Identical to: fun(1, 2, 3, y=4, z=5)
# x=1, args=(2, 3), y=4, kwargs={"z": 5}
```

Default Parameters/Argument Values

- Parameters can have optional **default argument values**:

```
def create_filled_list(size=3, val=0):
    return [val] * size
```

Default Parameters/Argument Values

- Parameters can have optional **default argument values**:

  ```
  def create_filled_list(size=3, val=0):
      return [val] * size
  ```

- Normal parameters must not appear afterwards, e.g., `(size=3, val=0, normal)`

Default Parameters/Argument Values

- Parameters can have optional **default argument values**:

```
def create_filled_list(size=3, val=0):
    return [val] * size
```

- Normal parameters must not appear afterwards, e.g.,

  `(size=3, val=0, normal)`

- Different ways of calling such a function:

```
create_filled_list()                # size=3, val=0
create_filled_list(2)               # size=2, val=0
create_filled_list(2, 1)            # size=2, val=1
create_filled_list(size=2)          # size=2, val=0
create_filled_list(val=1)           # size=3, val=1
create_filled_list(val=1, size=2) # size=2, val=1
create_filled_list(2, val=1)        # size=2, val=1
```

Default Parameters/Argument Values

- Parameters can have optional **default argument values**:

```
 def create_filled_list(size=3, val=0):
     return [val] * size
```

- Normal parameters must not appear afterwards, e.g.,

```
(size=3, val=0, normal)
```

- Different ways of calling such a function:

```
create_filled_list()                # size=3, val=0
create_filled_list(2)               # size=2, val=0
create_filled_list(2, 1)            # size=2, val=1
create_filled_list(size=2)          # size=2, val=0
create_filled_list(val=1)           # size=3, val=1
create_filled_list(val=1, size=2) # size=2, val=1
create_filled_list(2, val=1)        # size=2, val=1
```

- Default parameters are evaluated once in the beginning, so **mutable parameters** (e.g., lists) **can be changed**!

Type Hinting

- Parameters and return values can have **type hints**

Type Hinting

- Parameters and return values can have **type hints**
- Helpful to show what your function expects as inputs and returns as output

Type Hinting

- Parameters and return values can have **type hints**
- Helpful to show what your function expects as inputs and returns as output
- Example:

```
def get_max(x: int, y: int) -> int:
    return x if x > y else y
```

Type Hinting

- Parameters and return values can have **type hints**
- Helpful to show what your function expects as inputs and returns as output
- Example:

```
def get_max(x: int, y: int) -> int:
    return x if x > y else y
```

- Only hints, you can still pass and return anything!

Return Value

- Functions can return arbitrary values:

```
def get_max(x, y):
    return x if x > y else y
```

Return Value

- Functions can return arbitrary values:

```
def get_max(x, y):
    return x if x > y else y
```

- Or they can return nothing:

```
def print_hello():
    print("hello")
```

Return Value

- Functions can return arbitrary values:

```python
def get_max(x, y):
    return x if x > y else y
```

- Or they can return nothing:

```python
def print_hello():
    print("hello")
```

- In the background, Python actually still returns the special value **None**, so we could also write either of these two:

```python
def print_hello():
    print("hello")
    return None
```

```python
def print_hello():
    print("hello")
    return
```

Return Value

- Functions can return arbitrary values:

```
def get_max(x, y):
    return x if x > y else y
```

- Or they can return nothing:

```
def print_hello():
    print("hello")
```

- In the background, Python actually still returns the special value **None**, so we could also write either of these two:

```
def print_hello():
    print("hello")
    return None
```

```
def print_hello():
    print("hello")
    return
```

- Once the **return** keyword is encountered, the function will terminate its execution and return the specified value

NAMESPACES AND SCOPES

Namespaces and Scopes (1)

- A **namespace** is a dictionary that maps names (variables, functions, etc.) to their objects

Namespaces and Scopes (1)

- A **namespace** is a dictionary that maps names (variables, functions, etc.) to their objects
- There are several namespaces: the built-in namespace, the global/module namespace, enclosing/nested namespaces and the local namespace

Namespaces and Scopes (1)

- A **namespace** is a dictionary that maps names (variables, functions, etc.) to their objects
- There are several namespaces: the built-in namespace, the global/module namespace, enclosing/nested namespaces and the local namespace
- The **scope** of a name defines the namespace look-up order (it essentially determines the visibility of a name): First, the local namespace is searched, then any enclosing namespaces, followed by the global namespace and lastly the built-in namespace

Namespaces and Scopes (2)

- The **built-in** namespace contains all Python built-ins, like `len`, `dict`, `print`, etc.

Namespaces and Scopes (2)

- The **built-in** namespace contains all Python built-ins, like `len`, `dict`, `print`, etc.
- The **global** namespace contains names defined in the main script/module (e.g., global variables)

Namespaces and Scopes (2)

- The **built-in** namespace contains all Python built-ins, like `len`, `dict`, `print`, etc.
- The **global** namespace contains names defined in the main script/module (e.g., global variables)
- The **local** namespace contains names defined at the innermost level (e.g., local variables within a function)

Namespaces and Scopes (2)

- The **built-in** namespace contains all Python built-ins, like `len`, `dict`, `print`, etc.
- The **global** namespace contains names defined in the main script/module (e.g., global variables)
- The **local** namespace contains names defined at the innermost level (e.g., local variables within a function)
- In case of nested structures (e.g., a function within a function), the **enclosing** namespaces contain the names defined in the respective nesting level

Example

```
x = str(12)

def func(a)
    c = 10
    return a + c
```

- Relevant namespaces:
  - `str` is part of the built-in namespace
  - `x` and `func` are part of the global namespace
  - `a` and `c` are part of the local namespace
- For more details, such as using global variables in functions and the implications thereof, see the accompanying code file