Programming in Python

Functions, recursion, generators, lambda expressions
lecture 2

Department of Cybernetics and Artificial Intelligence
Technical University of Košice
Ing. Ján Magyar, PhD.

Function

- independent sequence of instructions within a program
- usually represents the execution of a task
- is called when we need to execute the given task
- has clearly defined input and output

Advantages of functions

- problem decomposition
  - “divide and conquer” approach
  - we first write solutions for simpler tasks, then join them
- abstraction
  - functions hide implementation details
  - the programmer knows only what input the function expects and what it returns
- lower development costs
- eliminating repeating code
- simple code maintenance

Function structure

- name
- parameters
- variables
- return values
- specification

Defining a function in Python

```
def name(parameters):
    body
    return return_value
```

Function parameters in Python

- we don’t need to define the type

  ```
  def square(number):
  ```

- we can assign default values

  ```
  def square(number, print_result=False):
  ```

  - parameters with a default value are optional when calling the function
- we can define functions with an unknown number of parameters

  ```
  def get_last(*keys):
  ```

  - we can work with parameters as a tuple
- when calling the function we can give the names of parameters, that way we don’t have to keep their order

Type aliases

- Python is a dynamically typed language, but we can indicate the expected data types

```
def square(number:int) -> int:
    return number ** 2
```

- for more complex types we can use type aliases:

```
ListResult = list[int]
DictionaryResult = dict[str, ListResult]
```

- aliases are used only for static checking, not during runtime

Defensive programming

- when implementing a function, we must consider that the function will also have to process invalid input
- before executing the task itself, we check the correctness and validity of input values
  - type check
  - value validity check (if limitations exist)
- output check before finishing a function is recommended

Variables in functions

- each function has access to global variables
- each function has a local namespace, which is invisible to other functions
- each function call will create its own namespace

Return values

- defined using the `return` keyword
- in Python each function has a return value – default is `None`
- a function can have multiple return values

```
return value1, value2, …
```

- we can store return values through assignment when calling a function

```
value1 = function1()
value1, value2 = function2()
```

Function specification

- written for the programmer/user
- part of the documentation
- describes the function aim, expected input and return value, as well as side effects
- syntax: in triple quotation marks on the first line of the function

Built-in Python functions

- type functions
- operation functions
- input-output functions

Type functions in Python

- `bool([x])`
- `chr(i)`
- `complex([real[, imag]])`
- `dict()`
- `enumerate(iterable, start=0)`
- `float([x])`
- `frozenset([iterable])`
- `int([x])`
- `isinstance(object, classinfo)`
- `list([iterable])`
- `set([iterable])`
- `str(object)`
- `tuple([iterable])`
- `type(object)`

Operation functions in Python

- abs(x)
- all(iterable)
- any(iterable)
- divmod(a, b)
- eval(expression, globals=None, locals=None)
- filter(function, iterable)
- hash(object)
- len(s)
- map(function, iterable, …)
- max(iterable)
- min(iterable)
- next(iterator)
- range(start, stop[, step])
- reversed(seq)
- round(number[, ndigits])
- sorted(iterable, key=None, reverse=False)
- sum(iterable[, start])
- zip(*iterables)

Input-output functions in Python

- `format(value[, format_spec])`
- `input([prompt])`
- `open(file, mode='r', encoding=None, …)`
- `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`

Working with files – opening and creating

```
open(file, mode='r', encoding=None, errors=None,
newline=None)
```

- `file` – path to file (if non-existent, it is created)
- `mode` – mode of working with the file (text or bytes)
  - `r` – read
  - `w` – write
  - `a` – append
  - `+` – update
  - `b/t` – binary/text mode
- `encoding` – only for texts
- `errors` – how to deal with errors
- `newline` – character used for newlines

Working with files – reading from the file

`file.read() / file.read(5)`

- reads all characters when no parameter is given
- the parameter defines the number of characters read

`file.readline() / file.readline(5)`

- reading from file by lines
- the parameter defines the number of lines read

`file.readlines()`

- reads all lines from the file as a list of strings

Working with files – writing

`write(string)`

- writes the string into the buffer

`writelines(list)`

- writes a list of strings into the buffer
- does not add newline character

`flush()`

- writes the buffer into the file

Working with files – closing the file

```
close()
```

- closes the file
- must always be called
- the buffer is written into the file

Working with files

```
my_file = open('path', 'r')

lines = my_file.readlines()

my_file.close()
```

```
with open('path', 'r') as my_file:
    lines = my_file.readlines()
```

calls `close()` explicitly

Main function

```
if __name__ == '__main__':
    main()
```

Best practice:

- always define `main()`
- `main()` should have as few calls as possible
- the body of the `if __name__` condition should be only the `main()` call, or processing input parameters

Passing parameters to the main function

- values available in `sys.argv`
- more elegant solution – using the `argparse` library

```
parser = argparse.ArgumentParser()
parser.add_argument('--path', metavar='path', required=True, help='path
to project folder')
parser.add_argument('--save', metavar='path', required=True, help='path
to save directory')
args = parser.parse_args()
main(workspace=args.path, schema=args.save)
```

Recursive functions

- function defined by calling themselves
- the definition has two parts
    - base case
        - simplest possible use case
        - statically defined value
    - inductive/recursive step
        - defines the calculation for more complex use cases
        - calls the function being defined

Recursion and iterative solutions

- recursion and loops are semantically equivalent
- some problems are easier to solved with one or the other approach
- if you can derive the calculation using simpler use cases, use recursion
- for brute-force algorithms, use iteration

- typical use cases for recursion: palindromes, Fibonacci numbers

Palindromes

A palindrome is a sequence of symbols that yields the same result when reading from left to right or right to left, e.g. 1001001.

Solving using recursion:

- an empty sequence is a palindrome
- a sequence with one symbol is a palindrome
- a sequence is a palindrome, if the first and last symbols are the same, and the middle is a palindrome

Fibonacci numbers

Define a sequence in which every member is the sum of the two previous members.

Solving using recursion:

- the first member is 0
- the second member is 1
- further members are the sum of the two previous ones

Brute force algorithm - barnyard problem

We have chicken and rabbits in a yard. Together they have 48 heads and 128 legs. How many are there of each?

- formally we can solve the problem as linear equations

  $x + y = 48$

  $2 * x + 4 * y = 128$
- for a computer, it is easier to solve the problem using brute force, testing each possible configuration until we find a solution

Fibonacci numbers – generators

- generators are special functions that work as iterators
- more effective work with memory (lazy evaluation)
- using **`yield`** instead of **`return`**
- the result is an iterator, which we can use in a **`for`** loop
- using **`next`** we get the next member of a sequence

Using generators

- working with large files
- processing big data
- generating infinite sequences
- pipelines

Lambda expression

- nameless function
- the body is a single expression defining the return value
- the definition must fit on a single line
- can have an arbitrary number of parameters

Defining lambda expressions

-
  ```
  lambda parameters: expression
  lambda x: 3 * x + 2
  ```

- can be stored in a variable and then called using the variable
  ```
  f = lambda x: 3 * x + 2
  f(2)
  ```
- today such use is not recommended, lambda expressions should be used only in a limited number of cases

Use cases for lambda expressions

- key for filtering
- key for sorting
- sorting complex data types (tuples, dictionary, objects)
- simple functions, which we do not use a large number of times

Conclusion

- function definition
- structure and function parts
- selected built-in functions
- recursion
- generators
- lambda expressions and their use