Compendio final certificación Python

Exam block #1: Computer Programming and Python Fundamentals

Study Pages

Objectives covered by the block:

PCEP 1.1 Understand fundamental terms and definitions

- interpreting and the interpreter, compilation and the compiler, lexis, syntax and semantics

PCEP 1.2 Understand Python's logic and structure

- keywords, instructions, indenting, comments

PCEP 1.3 Introduce literals and variables into code and use different numeral systems

- Boolean, integers, floating-point numbers, scientific notation, strings, binary, octal, decimal, and hexadecimal numeral system, variables, naming conventions, implementing PEP-8 recommendations

PCEP 1.4 Choose operators and data types adequate to the problem

- numeric operators: `**`, `*`, `/`, `%`, `//`, `+`, `–`, string operators: `*`, `+`, assignments and shortcut operators, unary and binary operators, priorities and binding, bitwise operators: `~`, `&`, `^`, `|`, `<<`, `>>`, Boolean operators: `not`, `and`, `or`, Boolean expressions, relational operators (`==`, `!=`, `>`, `>=`, `<`, `<=`), the accuracy of floating-point numbers, type casting

PCEP 1.5 Perform Input/Output console operations

- functions: `print()`, `input()`, `sep=` and `end=` keyword parameters, functions: `int()` and `float()`

Understanding fundamental concepts

1. A **language** is a means (and a tool) for expressing and recording thoughts.

2. A **natural language** is a language people use to communicate with each other in everyday life. English, Russian, German, Swahili, and Hindi are examples of natural languages.

3. A **programming language** is a language developed by humans and used to communicate with computers. A programming language has a set of means to instruct a computer what to do and how.

4. A **high-level programming language** is a programming language which operates on a high level of abstraction thereby allowing the developer to ignore the physical details of the computer's hardware, for example the CPU type, memory size and organization, etc. Python, JavaScript, and C/C++ are all examples of high-level programming languages.

5. A **machine language** is a language placed at the lowest level of computer programming. It's a sequence of bits (binary digits usually recognized as **0** and **1**) which directly forces the CPU to execute the desired elementary operations.

6. An **instruction list** (abbreviated to **IL**) is a list of all elementary (atomic) operations which can be executed by a certain CPU. For example, **x86** (used in personal computers) and **arm** (used in mobile devices) processors have different and incompatible instruction lists).

7. A **source code** is a text encoded in any of the programming languages (regardless of the language's level). Usually, the source code is put inside a text file which resides inside the developer's computer filesystem, while the file name's extension reveals the programming language used to write the code (for example, files with names which ends with `.py` contain Python source code, while the `.cpp` extension marks files which hold C++ (usually pronounced as *see-plus-plus*) source code.

8. Any language (no matter if it's natural or artificial) is constituted by:
    - an **alphabet** understood as a set of symbols used to build words of a certain language (e.g. the Latin alphabet for English, the Cyrillic alphabet for Russian, Kanji for Japanese, and so on).
    - a **lexis**, also known as a **dictionary**, is a set of words the language offers its users (for example, the word "chat" is present both in English and French dictionaries, but its meaning is obviously different).
    - **syntax** is a set of rules used to determine if a certain sequence of words forms a valid sentence.
    - **semantics** is defined as a set of rules which settle whether or not a certain phrase or sentence makes sense in a given language.

9. A source code **cannot be directly executed by a computer**. To make it possible the source code has to be **translated** into a machine code accepted by a target computer and its CPU. This task can be done using two different techniques:
    - **compilation** performed by a one-time translation of the source program; an executable binary file is created in effect – the file can be run at any time without the need to have the source code; the program that performs the above translation is called a **compiler** or **translator**.
    - **interpretation** which involves a dedicated program designed to translate the source program on-the-fly each time it has to be run; the program performing this task is called an **interpreter**; this means that the interpreter is needed whenever the source code has to be executed.

10. A specific programming language is designed to be the object of either compilation or interpretation (this choice imposes certain distinctive features onto the language). For example, **Python** is an **interpreted** programming language, while **C++** is a compiled one.

11. The interpreter and its environment, created and distributed by the **Python Software Foundation** (PSF) is written mostly in the C programming language. It allows Python to be easily ported and migrated to all platforms providing the ability to compile and run C language programs. This is also why the PSF implementation is often referred to as **CPython**. CPython is the most widely used implementation of Python.

Python basic types and literals

1. A **literal** is data whose value is **determined by the literal itself**. Different kinds of data are coded in different ways, enabling Python (and the human reader of the source code) to determine each literal's **type**. Knowing each argument's type is crucial to understand what operations are legal, and what results can be returned.

2. **Integer** (`int` for short) is a type dedicated to storing **integral numbers**, that is, numbers that lack fractional parts. For example, 1 is an integral number and 1.5 is not.

3. Most used integer literals in Python consist of a sequence of **decimal digits**. Such a sequence cannot include white spaces, but can contain any number of _ (underscore) characters. Note: there must not be more than one underscore between two digits, and the underscore must be neither the last nor the first character of the literal. Underscores don't change a literal's value, and their only role is to improve literal readability. Integer literals can be preceded by any number of `-` (minus) or `+` (plus) characters, which are applied to set the literal's sign.

4. Here are some examples of correct integer literals:
   - `1_111` and `1111` encode the same integer value (*one thousand one hundred and eleven*)
   - `-+-3` and `-3` denote the same integer value (*minus three*)
   - `+1` and `1` encode the same integer value (*one*)

5. Integer literals may be written using radices other than 10:
   - if a literal starts with either a `0o` or `0O` digraph, it's an **octal** value (note: it must contain octal digits only!)
   - `0o10` encodes an integer value equal to *eight*.
   - if a literal starts with either a `0x` or `0X` digraph, it's a hexadecimal value (note: letters from **a** to **f** used as hexadecimal digits may be upper- or lower-case)
   - `0X11` encodes an integer value equal to *seventeen*.
   - if a literal starts with either a `0b` or `0B` digraph, it's a binary value (note: it must contain **0**s and **1**s only!)
   - 0b111 encodes an integer value equal to seven.

6. **Floating point** (`float` for short) is a type designed to store **real** numbers (in the mathematical sense), that is, numbers whose decimal expansion is or can be non-zero. Such a class of numbers includes fractions (integers don't).

7. Float literals are distinguished from integers by the fact that they contain a **dot** (`.`) or the letter *e* (lower- or upper-case) or both. If the only digits which exist before or after the dot are zeros, they can be omitted. Like integers, floats can be preceded by any number of `-` and `+` characters, which determine the number's sign. White spaces are not allowed, while underscores are.

8. If the float literal contains the letter *e*, it means that its value is **scaled**, that is, it's multiplied by a **power of 10** while the exponent (which must be an integer!) is placed directly after the letter. A record like this:

   m**E**n

   is treated as a value equal to:

   m × 10n

   This syntax is called *scientific notation* and is used to denote a number whose absolute value is extremely large (close to infinity) or extremely small (close to zero).

9. Here are some examples of correct float literals:
   - `1.1` – *one and one-tenth*
   - `1.0` (`1.` for short) – *one point zero*
   - `0.1` (`.1` for short) – *one-tenth*
   - `1E1` – *ten point zero*
   - `1e-1` – *one-tenth*
   - `-1.1E-1` – *minus eleven hundredths*

Python basic types and literals: continued

10. **String** literals are sequences (including empty ones) of characters (digits, letters, punctuation marks, etc.). There are two kinds of string literal:

  - **single-line**, when the string itself begins and ends in the same line of code: these literals are enclosed in a pair of `'` (apostrophe) or `"` (quote) marks.
  - **multi-line**, when the string may extend to more than one line of code: these literals are enclosed in a pair of trigraphs either `"""` or `'''`
  - strings enclosed inside apostrophes can contain quotes, and vice versa.
  - if you need to put an apostrophe inside an apostrophe-limited string, or a quote inside a quote-limited string, you must precede them with the `\` (backslash) sign, which acts as an escape character (a character which changes the meaning of the character that follows it); some of the most used escape sequences are:
    - `\\` – backslash
    - `\'` – apostrophe
    - `\"` – quote
    - `\n` – newline character
    - `\r` – carriage return character
    - `\t` – horizontal tab character

11. Here are some examples of correct string literals:
    - `"Hello world"`
    - `'Goodbye!'`
    - `''` (an empty string)
    - `"Python's den"`
    - `'Python\'s den'`
    - `"""Two lines"""`

12. **Boolean literals** denote **the only two possible values** used by the *Boolean algebra* – their only acceptable denotations are `True` and `False`.

13. The `None` literal denotes an **empty value** and can be used to indicate that a certain item contains no usable value.

Operators

1. An operator is a symbol that **determines an operation to perform**. The operator along with its arguments (operands) forms an expression that is subject to evaluation and provides a result.

2. There are three basic groups of operators in Python:
   - arithmetic, whose arguments are numbers;
   - string, which operates on strings or strings and numbers;
   - Boolean, which expects that their arguments are Boolean values.

3. A unary operator is an operator with only one operand.

4. A binary operator is an operator with two operands.

Unary arithmetic operators

Operator	Meaning	Example
`-`	Change argument's sign	`-(-2)` is equal to `2`
`+`	Preserve argument's sign	`+(-2)` is equal to `-2`

Binary arithmetic operators (ordered according to descending priority)

Priority	Operator	Name	Example	Meaning	Result	Result Type
Highest	`**`	Exponentiation	`2 ** 3`	2^3	`8`	• `int` if both arguments are ints • `float` otherwise
•	`*`	Multiplication	`2 * 3`	2×3	`6`	• `int` if both arguments are ints • `float` otherwise
•	`/`	Division	`4 / 2`	$4 \div 2$	`2.0`	• always `float` • raises `ZeroDivisionError` when divider is zero
•	`//`	Integer division	`5 // 2`		`2`	• `int` if both arguments are ints • `float` otherwise • raises `ZeroDivisionError` when divider is zero

•	`%`	Remainder (modulo)	`5 % 2`	5 mod 2	`1`	• `int` if both arguments are ints • `float` otherwise • raises `ZeroDivisionError` when divider is zero
	`+`	Addition	`2 + 1`	2 + 1	`3`	• `int` if both arguments are ints • `float` otherwise
Lowest	`–`	Subtraction	`2 - 1`	2 – 1	`1`	• `int` if both arguments are ints • `float` otherwise

- pairs of parentheses can be used to change the order of operations, for example:
  - `2 + 3 * 4` evaluates to `14`
  - `(2 + 3) * 4` evaluates to `20`

- when operators of the same priority (other than `**`)are placed side-by-side in the same expression, they are evaluated **from left to right**: therefore, the expression:

  ```
  1 / 2 * 2
  ```

  evaluates to `1.0`, not to `0.25`.

  This convention is called **left-sided binding**.

- when more than one `**` operator is placed side-by-side in the same expression, they are evaluated **from right to left**: therefore, the expression:

  ```
  2 ** 2 ** 3
  ```

  evaluates to `256` (2^8), not to `64` (4^3) – this is **right-sided binding**.

String operators (ordered according to descending priorities)

The rules governing the use of operators and parentheses remain the same, and left-sided binding is in effect.

Priority	Operator	Name	Example	Result	Result type

Highest	`*`	Replication	`'a' * 3`
			`3 * 'a'`
				`'aaa'`	Always a string

Middle	`+`	Concatenation	`'a' + 'z'`
			`'z' + 'a'`
				`'az'`
				`'za'`
					Always a string

Boolean operators (ordered according to descending priorities)

Boolean operators demand Boolean arguments, and always result in a Boolean result. The rules governing the use of operators and parentheses remain the same, including left-sided binding.

Priority	Operator	Name	Example	Result
Highest	`not`	negation	`not false`	`True`
			`not True`	`False`
Middle	`and`	conjunction	`False and False`	`False`
			`False and True`	`False`
			`True and False`	`False`
			`True and True`	`True`
Lowest	`or`	disjunction	`False or False`	`False`
			`False or True`	`True`
			`True or False`	`True`
			`True or True`	`True`

Relational operators

Relational operators compare their arguments to diagnose the relationship between them, and always return a Boolean value indicating the comparison result.

Operator	Name	Example	Result

`==`	equal to	`2 == 1 False`
`!=`	not equal to	`2 != 1 True`
`>`	greater than	`2 > 1  True`

`>=`	greater or equal	`2 >= 1 True`
		`1 >= 1 True`

`<`	less than	`2 < 1  False`

`<=`	less or equal	`2 <= 1 False`
		`1 <= 1 True`

Variables

1. A variable is **a named container able to store data**.

2. A variable's name can consist of:
    - **letters** (including non-Latin ones)
    - **digits**
    - **underscores** (_)

   and **must start with a letter** (note: underscores count as letters). Upper- and lower-case letters are treated as different.

3. Variable names which start with underscores play a specific role in Python – don't use them unless you know what you're doing.

4. Variable names are not limited in length.

5. The name of the variable must not be any of Python's **keywords** (also known as **reserved words**). The complete list of Python 3.8 keywords looks as follows:

```
'False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await',
'break', 'class', 'continue', 'def', 'del', 'elif', 'else',
'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in',
'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return',
'try', 'while', 'with', 'yield'
```

* Note: Python 3.9 introduced a new keyword, `__peg_parser__`, which is an easter egg related to the rollout of a new PEG parser for CPython. You can read more about this in PEP 617. The keyword will most probably be removed in Python 3.10.

6. These are some legal, but not necessarily the best, Python variable names:
   - `i`
   - `counter_2`
   - `_`
   - `Tax22`

The assignment operator

7. The assignment operator = is designed to assign a value to a variable:

```
variable = expression
```
For example:

```
counter = 0

pi2 = 3.1415 ** 2
```

or – when more than one variable is assigned with the same value:

```
variable_1 = variable_2 = ... = expression
```
For example:

```
counter = stages = 0
```

8. A variable must be **assigned (defined)** before its first use – using a variable without prior definition (assignment) raises the *NameError* exception.

9. Python is a **dynamically typed** language, which means that a variable can freely change its type according to the last assignment it took part in.

10. There are some **short-cut (compound) operators** which simplify certain kinds of assignments.

Compound operators

Compound Arithmetic Operators

Operator	Meaning	Example
`variable += expression`	`variable = variable + (expression)`	`counter += 1`
`variable -= expression`	`variable = variable - (expression)`	`due -= ret`
`variable *= expression`	`variable = variable * (expression)`	`next_power *= next_power`
`variable /= expression`	`variable = variable / (expression)`	`fraction /= fraction`
`variable //= expression`	`variable = variable // (expression)`	`always_one //= always_one`
`variable %= expression`	`variable = variable % (expression)`	`always_zero %= 1`
`variable **= expression`	`variable = variable ** (expression)`	`square_me **= 2`

Compound String Operators

Operator	Meaning	Example
`variable += expression`	`variable = variable + (expression)`	`header += '****'`
`variable *= expression`	`variable = variable * (expression)`	`doubled *= 2`

Note: Boolean operators have no short-cut variants.

Comments

11. A part of the code line which starts with hash (`#`), which is not a part of a string literal, is considered a **comment** (a part of the code which is ignored by the interpreter)

    For example, these two lines contain comments:

```
# This is a line which is completely ignored by the Python interpreter.
result = True # only part of this line is a comment
```

Basic Input/Output

1. The `input()` function is used to interact with the user, allowing them to suspend program execution and to enter a sequence (including an empty one) of characters, which is the function's **result**.

2. If the user presses `<Enter>` while providing no other input, the `input()` function returns an empty string as a result.

   For example, the `name` variable is assigned with a string inputted by the user:

   ```
   name = input()
   ```

3. The `input()` function accepts an optional argument which is a string; the argument is printed on the screen before the actual input begins.

   For example, the user is prompted to input the name:

   ```
   name = input('What is your name?')
   ```

4. To convert the user's input into an integer number, the `int()` function can be used.

5. An operation during which the data changes its type is called **type casting**.

For example, the user is asked to input a year of birth. The input is converted into an integer number and assigned to the `b_year` variable:

```
b_year = int(input('What is your year of birth?'))
```

6. To convert the user's input into a float number, the `float()` function can be used.

   For example, the user is asked to input a height measured in meters. The input is converted into a float number and assigned to `the m_stature` variable:

   ```
   m_stature = float(input('What is your height in meters?'))
   ```

7. If the `int()` or `float()` is not able to perform the conversion due to the incorrect contents of the user's input, the `ValueError` exception is raised.

8. Unless otherwise specified, the printed values are separated by single spaces.

   For example, the following snippet sends `1 2 3` to the screen:

   ```
   print(1, 2, 3)
   ```

9. If the invocation makes use of the `sep` keyword parameter, the parameter's value (even when it's empty) is used to separate outputted values instead of spaces.

   For example, the following snippet sends `1*2*3` to the screen:

   ```
   print(1, 2, 3, sep='*')
   ```

10. Unless otherwise specified, each `print()` function invocation sends the newline character to the screen before its completion, therefore each `print()` function's output starts on a new line.

For example, the following snippet produces two lines of output on the screen:

```
print('Alpha')

print('Bravo')
```

11. If the invocation makes use of the `end` keyword parameter, the parameter's value is (even when it's empty) used to complete the output instead of the newline.

For example, the following snippet produces one line of output on the screen:

```
print('Alpha', end='')

print('Bravo')
```

12. The `end` and `sep` parameters can be used together.

For example, the following snippet produces one line of asterisk-separated letters:

```
print('A', 'B', sep='*', end='*')

print('C')
```

Exam block #2: Control Flow – Conditional Blocks and Loops

Study Pages

Objectives covered by the block:

PCEP 2.1 Make decisions and branch the flow with the *if* instruction

- conditional statements: *if*, *if-else*, *if-elif*, *if-elif-else*,
- multiple conditional statements,
- nesting conditional statements.

PCEP 2.2 Perform different types of iterations

- the pass instruction,
- building loops with *while*, *for*, *range()*, and *in*,
- iterating through sequences, expanding loops with *while-else* and *for-else*,
- nesting loops and conditional statements,
- controlling loop execution with *break* and *continue*.

Conditional statements

1. **The conditional statement** (the `if` statement) is a means allowing the programmer to branch the execution path and to execute (or not) selected instructions when a certain condition is met (or not).

2. The basic form of the `if` statement looks as follows:

```
if condition:

instructions
```

3. The `condition` is an **expression** – if it evaluates to `True`, or to a **non-zero numeric value**, or to a **non-empty string** and is not `None`, it is fulfilled (met), and the nested instructions placed after the `if` are **executed**.

4. When the condition is not met, these instructions are **skipped**.

5. When there is only one instruction that should be executed conditionally, the instruction can be written in the following form:

```
if condition: instruction
```

6. For example, the following snippet prints `TRUE` to the screen:

```
counter = 1

if counter > 0:

print('TRUE')
```

7. The empty instruction denoted by the `pass` keyword can be used to indicate that no action should be performed in the specific context. As the `if` instruction syntax

insists that there should be at least one statement after it, the following snippet does not affect program execution:

```
if condition:

pass
```

8. It is suggested to use one tabulation character to make one indent level in Python code, while the recommended tab size (settable in virtually all code editors) is 4.

9. The `else` branch can be used to specify a part of the code that should be executed when the condition is not met:

```
if condition:

instructions

else:

instructions
```

10. For example, the following snippet prints `TRUE` when the `counter` variable is greater than zero, and `FALSE` otherwise:

```
if counter > 0:

print('TRUE')

else:

print('FALSE')
```

11. To check more than one condition within one conditional block, the `elif` branch or branches may be employed. In that case, not more than one `if/elif` branch can be executed. The `else` branch is optional, and must be the last branch.

12. For example, the following snippet prints `PLUS` when the counter variable is greater than zero, `MINUS` when it's less than zero, and `ZERO` when it's equal to zero:

```
if counter > 0:

print('PLUS')

elif counter < 0:

print('MINUS')

else:

print('ZERO')
```

Loops

1. **The `while` loop statement** is a means allowing the programmer to **repeat** the execution of the selected part of the code as long the specified condition is **true**. The condition is checked before the loop's first turn, and therefore the loop's **body** may not even be executed once.

2. The basic form of the while statement looks as follows:

```
while condition:

instructions
```

3. The `condition` is an **expression** – as long it evaluates to `True`, or to a **non-zero** numeric value, or to a **non-empty** string, it is fulfilled (met) and is not `None`, the nested instructions placed after the `while` are **executed**.

4. When the condition is **not met**, these instructions are **skipped**.

   For example, the following snippet prints `TRUE` twice to the screen:

```
counter = 2
```

```
if counter > 0:

print('TRUE')

counter -= 1
```

5. The `else` branch can be used to specify a part of the code that should be executed when the loop’s condition is not met:

   ```
   while condition:

   instructions

   else:

   instructions
   ```

   For example, the following snippet prints `TRUE FALSE` to the screen:

   ```
   counter = 1

   while counter > 0:

   print('TRUE', end=' ')

   counter -= 1

   else:

   print('FALSE')
   ```

6. If the condition is met at the beginning of the loop and there is no chance that the condition value has changed inside the body of the loop, the execution enters an **infinite loop** which cannot be broken without the user's intervention, for example by pressing the Ctrl-C (Ctrl-Break) key combination.

   For example, the following snippet infinitely prints `TRUE` to the screen:

   ```
   while True:
   ```

```
print('TRUE', end=' ')
```

7. The `for` loop statement is a means allowing the programmer to **repeat** the execution of the selected part of the code when the **number of repetitions can be determined in advance**. The `for` statement uses a dedicated variable called a **control variable**, whose subsequent values reflect the status of the iteration.

8. The basic form of the for statement looks as follows:

```
for control_variable in range(from, to, step):

instructions
```

9. The `range()` function is a generator responsible for the creation of a series of values starting from `from` and ending before reaching `to`, incrementing the current value by `step`.

10. The invocation `range(i,j)` is the equivalent of `range(i, j, 1)`

11. The invocation `range(i)` is the equivalent of `range(0, i)`

For example, the following snippet prints `0,1,2,` to the screen:

```
for i in range(3):

print(i, end=',')
```

For example, the following snippet prints `2 1 0` to the screen:

```
for i in range(2, -1, -1):
```

```
print(i, end=' ')
```

12. The `else` branch can be used to specify a part of the code that should be executed when the loop's body is **not entered**, which may happen when the range being iterated is **empty** or when all the range's values have already been **consumed**.

    For example, the following snippet prints `0 1 2 FINISHED` to the screen:

```
for i in range(3):

print(i, end=' ')

else:

print('FINISHED')
```

    For example, the following snippet prints `FINISHED` to the screen:

```
for i in range(1,1):

print(i, end=' ')

else:

print('FINISHED')
```

13. **The `break` statement** can be used inside the loop's body only, and causes immediate **termination** of the loop's code. If the loop is equipped with the `else` branch, it is omitted.

    For example, these two snippets print `0 1` to the screen:

```
# break inside for

for i in range(3):
```

```
if i == 2:

break

print(i, end=' ')

else:

print('FINISHED')


# break inside while

i = 1

while True:

print(i, end=' ')

i += 1

if i == 3:

break

else:

print('FINISHED')
```

14. **The `continue` statement** can be used inside the loop's body only, and causes an immediate **transition** to the next iteration of the for loop, or to the `while` loop's condition check.

For example, these two snippets print `0 2 FINISHED` to the screen:

```
# continue inside for

for i in range(4):

if i % 2 == 1:

continue

print(i, end=' ')
```

```
else:

print('FINISHED')


# continue inside while

i = -1

while i < 3:

i += 1

if i % 2 != 0:

continue

print(i, end=' ')

else:

print('FINISHED')
```

Exam block #3: Data Collections – Tuples, Dictionaries, Lists, and Strings

Study Pages

Objectives covered by the block:

PCEP 3.1 Collect and process data using lists

- constructing vectors, indexing and slicing, the *len()* function, basic list methods (*append()*, *insert()*, *index()*) and functions (*len()*, *sorted()*, etc.), the *del* instruction; iterating through lists with the *for* loop, initializing loops; *in* and *not in* operators, list comprehensions; copying and cloning, lists in lists: matrices and cubes.

PCEP 3.2 Collect and process data using tuples

- tuples: indexing, slicing, building, immutability; tuples vs. lists: similarities and differences, lists inside tuples and tuples inside lists.

PCEP 3.3 Collect and process data using dictionaries

- dictionaries: building, indexing, adding and removing keys; iterating through dictionaries and their keys and values, checking the existence of keys; *keys()*, *items()* and *values()* methods.

PCEP 3.4 Operate with strings

- constructing strings, indexing, slicing, immutability; escaping using the \ character; quotes and apostrophes inside strings, multi-line strings, basic string functions and methods.

Lists

1. A **list** is a data aggregate that contains a certain number (including zero) of elements of any type.

2. Lists are **sequences** – they can be iterated, and the order of the elements is established.

3. Lists are **mutable** – their contents may be changed.

4. Lists can be **initialized** with list literals. For example, these two assignments instantiate two lists – the former is empty, while the latter contains three elements:

```
empty_list = []

three_elements = [1, 'two', False]
```

5. The number of elements contained in the list can be determined by the `len()` function. For example, the following snippet prints `3` to the screen:

```
print(len(['a', 'b', 'c'])
```

6. Any of the list's elements can be accessed using **indexing**. List elements are indexed by integer numbers starting from **zero**. Therefore, the first list element's index is **0** while the last element's index is equal to the **list length minus 1**. Using indices that are not integers raises the `TypeError` exception. For example, the following snippet prints `a b c 0 1 2` to the screen:

```
the_list = ['a', 'b', 'c']

counter = 0

for ix in range(len(the_list)):

print(the_list[ix], end=' ')

the_list[ix] = counter
```

```
counter += 1

for ix in range(len(the_list)):

print(the_list[ix], end=' ')
```

7. The list elements can be indexed with **negative numbers**, too. In this case, **-1** accesses the last element of the list, and **-2** accesses the one before the last, and so on. The alternative first list element's index is `-len(list)`.

8. An attempt to access a non-existent list element (when the index goes out of the permissible range) raises the `IndexError` exception.

9. A **slice** is a means by which the programmer can create a new list using a part of the already existing list.

10. The most general slice looks as follows:

```
the_list[from:to:step]
```

and selects those elements whose indices start at `from`, don't exceed `to`, and change with `step`. For example, the following snippet prints `['b', 'd']` to the screen:

```
print((1,2,3)[4:5])
```

11. The following assumptions are made regarding the slices:
    - `the_list[from:to]` is equivalent to `the_list[from:to:1]`
    - `the_list[:to]` is equivalent to `the_list[0:to]`
    - `the_list[from:]` is equivalent to `the_list[from:len(the_list)-1]`
    - `the_list[:]` is equivalent to `the_list[0:len(the_list)-1]`

12. Slices – like indices – can take negative values. For example, the following snippet prints `[1,2]` to the screen:

```
the_list = [0, 1, 2, 3]

print(the_list[-3:-1])
```

Lists and strings

1. If any of the slice's indices exceeds the allowable range, **no exception is raised**, and the non-existent elements are not taken into consideration. Therefore, it is possible that the resulting slice is an **empty** list.

2. Assigning a list to a list **does not copy elements**. Such an assignment results in a situation when more than one name identifies the same data aggregate.

   For example, the following snippet prints `True` to the screen:

```
list_a = [1]

list_b = list_a

list_b[0] = 0

print(list_a[0] == list_b[0])
```

   As the slice is a copy of the source list, the following snippet prints `False` to the screen:

```
list_a = [1]

list_b = list_a[:]

list_b[0] = 0

print(list_a[0] == list_b[0])
```

3. The `.append(element)` method can be used to **append an element** to the end of an existing list. For example, the following snippet outputs `[1]` to the screen:

```
the_list = []

the_list.append(1)

print(the_list)
```

4. The `.insert(at_index, element)` method can be used to **insert** the element at the `at_index` of the existing list. For example, the following snippet outputs `[2, 1]` to the screen:

```
the_list = [1]

the_list.insert(0, 2)

print(the_list)
```

5. The `del the_list[index]` instruction can be used to **remove** any of the existing list elements. For example, the following snippet prints `[]` to the screen:

```
the_list = [1]

del the_list[0]

print(the_list)
```

6. The `in` and `not` in operators can check whether any value is contained inside the list or not. For example, the following snippet prints `True False` to the screen:

```
the_list = [1, 'a']

print('a' in the_list, 1 not in the_list)
```

7. Lists can be **iterated through** (traversed) by the `for` loop, which allows the programmer to scan all their elements without the use of explicit indexing. For example, the following snippet prints `1 2 3` to the screen:

```
the_list = [1,2,3]

for element in the_list:

print(element, end=' ')
```

8. **List comprehension** allows the programmer to construct lists in a compact way. For example, the following snippet prints `[1,2,3]` to the screen:

```
the_list = [x for x in range(1,4)]

print(the_list)
```

9. **Strings**, like lists, are sequences, and in many contexts they behave like lists, especially when they are indexed and sliced or are arguments of the `len()` function.

10. The `in` and `not` in operators can be applied to strings to check if any string is a **part of another string**. An empty string is considered a part of any string, including an empty one.

11. Strings are **immutable** and their contents cannot be changed.

Tuples

1. A **tuple**, like a list, is a data aggregate that contains a certain number (including zero) of elements of any type. Tuples, like lists, are **sequences**, but they are **immutable**. You're not allowed to change any of the tuple elements, or add a new element, or remove an existing element. Attempting to break this rule will raise the `TypeError` exception.

2. Tuples can be initialized with tuple literals. For example, these assignments instantiate three tuples – one empty, one one-element, and one two-element:

```
empty_tuple = () # tuple() has the same meaning

one_element_tuple = tuple(1) # must not be replaced with (1)!

one_element_tuple = 1, # the same effect as above

two_element_tuple = (1, 2.5)

two_element_tuple = 1, 2.5 # the same effect as above
```

3. The **number of elements** contained in the tuple can be determined by the `len()` function. For example, the following snippet prints `4` to the screen:

```
print(len((1, 2.2, '3', True))
```

   Note the inner pair of parentheses – they **cannot be omitted**, as it will cause the tuple to be replaced with four independent values and will cause an error.

4. Any of the tuple's elements can be accessed using indexing, which works in the same manner as in lists, including slicing.

5. An attempt to access a non-existent tuple element raises the `IndexError` exception.

6. If any of the slice's indices exceeds the permissible range, no exception is raised, and the non-existent elements are not taken into consideration. Therefore, the resulting slice may be an empty tuple. For example, the following snippet outputs `()` to the screen:

```
print((1,2,3)[4:5])
```

7. The `in` and `not` in operators can check whether or not any value is contained inside the tuple.

8. Tuples can be iterated through (traversed) by the `for` loop, like lists.

9. The `+` operator joins tuples together.

10. The `*` operator multiplies tuples, just like lists.

Dictionaries

1. A **dictionary** is a data aggregate that gathers `pairs of values`. The first element in each pair is called the key, and the second one is called the `value`. Both keys and values can be of any type.

2. Dictionaries are **mutable** but **are not sequences** – the order of pairs is imposed by the order in which the keys are entered into the dictionary.

3. Dictionaries can be initialized with dictionary literals. For example, these assignments instantiate two dictionaries – one empty and one containing two key:value pairs:

```
empty_dictionary = {}

phone_directory = {'Emergency': 911, 'Speaking Clock': 767}
```

Dictionaries – continued

4. Accessing a dictionary's value requires the use of its key. For example, the following line outputs `911` to the screen:

```
print(phone_directory['Emergency'])
```

5. An attempt to access an element whose key is absent in the dictionary raises the `KeyError` exception.

6. The `in` and `not` in operators can be used to check whether a certain key exists in the dictionary. For example, the following line prints `True False` to the screen:

```
print('Emergency' in phone_directory, 'White House' in
phone_directory)
```

7. The `len()` function returns the **number of pairs** contained in the directory. For example, the following line outputs `0` to the screen:

```
print(len(empty_directory))
```

8. **Changing** a value of the existing key is done by an **assignment**. For example, the following snippet outputs `False` to the screen:

```
attendance = {'Bob': True}

attendance['Bob'] = False

print(attendance['Bob'])
```

9. **Adding** a new pair to the dictionary resembles a regular assignment. For example, the following snippet outputs `2` to the screen:

```
domains = {'au': 'Australia'}

domains['at'] = 'Austria'

print(len(domains))
```

10. **Removing** a pair from a dictionary is done with the `del` instruction. For example, the following snippet outputs `0` to the screen:

```
currencies = {'USD': 'United States dollar'}

del currencies['USD']

print(len(currencies))
```

11. When iterated through by the for loop, the dictionary **displays only its keys**. For example, the following snippet outputs `A B` to the screen:

```
phonetic = {'A': 'Alpha', 'B': 'Bravo'}

for key in phonetic:

print(key, end=' ')
```

12. The `.keys()` method returns a **list of keys** contained in the dictionary. For example, the following snippet outputs `A B` to the screen:

```
phonetic = {'A': 'Alpha', 'B': 'Bravo'}

for key in phonetic.keys():

print(key, end=' ')
```

13. The `.values()` method returns a list of values contained in the dictionary. For example, the following snippet outputs `Alpha Bravo` to the screen:

```
phonetic = {'A': 'Alpha', 'B': 'Bravo'}

for value in phonetic.values():

print(value, end=' ')
```

14. The `.items()` method returns a **list of two-element** tuples, each filled with **key:value** pairs. For example, the following snippet outputs `('A', 'Alpha') ('B', 'Bravo')` to the screen:

```
phonetic = {'A': 'Alpha', 'B': 'Bravo'}

for item in phonetic.items():

print(item, end=' ')
```

Exam block #4: Functions and Exceptions

Study Pages

Objectives covered by the block:

PCEP 4.1 Decompose the code using functions

- defining and invoking user-defined functions and generators; the *return* keyword, returning results, the *None* keyword, recursion.

PCEP 4.2 Organize interaction between the function and its environment

- parameters vs. arguments; positional, keyword and mixed argument passing; default parameter values, name scopes, name hiding (shadowing), the *global* keyword.

PCEP 4.3 Python Built-In Exceptions Hierarchy

- *BaseException*, *Exception*, *SystemExit*, *KeyboardInterrupt*, abstract exceptions, *ArithmeticError*, *LookupError* along with *IndexError* and *KeyError*; *TypeError* and *ValueError* exceptions, the *AssertError* exception along with the *assert keyword*.

PCEP 4.4 Basics of Python Exception Handling

- *try-except*, *try-except* Exception, ordering the *except* branches, propagating exceptions through function boundaries; delegating responsibility for handling exceptions.

Functions

1. A **function** is a named, separate part of the code that can be **activated on demand**. A function can **perform an action**, or **return a result**, or **both**.

2. The simplest function, which does nothing and returns no result, can be defined in the following way:

```
def lazy():

pass
```

3. Activating a function is done by the **function invocation** (function call). The `lazy()` function defined above can be invoked by the following clause:

```
lazy()
```

4. Function definition must **precede its invocation**. Breaking this rule raises the `NameError` exception.

5. A function can be equipped with an arbitrary number of **parameters**. The parameters behave like variables known inside the function only, and their values are set during the invocation. The invocation must provide **as many arguments as needed to initialize all parameters**. Breaking this rule results in raising the `TypeError` exception.

6. If a function is supposed to evaluate a result, it must perform the `return` *expression* instruction, which immediately terminates function execution and causes the function to return the *expression* value to the invoker. If the function does not execute the instruction, or utilizes return without an expression, the None value is returned implicitly. For example, the following snippet prints `True None` to the screen:

```
def function(parameter):

if parameter == False:
```

```
return True

print(function(False), function(True))
```

7. A function definition can declare **default values** for some or all of its parameters. When the invocation does not provide arguments for these parameters, the default values are taken into consideration. Note: parameters with default values must not precede the ones without them. For example, the following snippet prints `True False` to the screen:

   ```
   def function(parameter = False):

   return parameter

   print(function(True), function())
   ```

8. The **positional** parameter passing technique is a technique based on the assumption that the arguments are associated with the parameters **based upon their position** (i.e. the first argument value goes to the first parameter, and so on) For example, the following snippet outputs `1 2 3` to the screen:

   ```
   def function(a, b, c):

   print(a, b, c)

   function(1, 2, 3)
   ```

9. The **keyword** parameter passing technique is a technique based on the assumption that the arguments are associated with the parameters based upon the parameter's **names**, which must be explicitly specified during the invocation. For example, the following snippet outputs `1 2 3` to the screen:

   ```
   def function(a, b, c):

   print(a, b, c)

   function(c=3, a=1, b=2)
   ```

Functions – continued

10. A function definition can declare **default values** for some or all of its parameters. When the invocation does not provide arguments for these parameters, the default values are taken into consideration. Note: parameters with default values must not precede the ones without them. For example, the following snippet prints `1 2 3` to the screen:

```
def function(a, b, c):

print(a, b, c)

function(1, c=3, b=2)
```

11. Note that the following invocation is incorrect and will raise the `TypeError` exception, because the a parameter is set twice (once with the positional passing and once with the keyword passing) while the `c` parameter is not set at all.

```
function(1, a=1, b=2)
```

12. A **scope** is the part of the code where a certain name is properly recognizable.

13. A variable existing outside a function has a scope which includes the function's bodies.

14. A variable defined inside the function has a scope inside the function's body only.

15. If a certain variable is used inside a function and the variable's name is listed as an argument of the `global` keyword, it has **global scope**, and it is also recognizable outside the function. For example, the following snippet outputs `2` to the screen:

```
def function():
```

```
global variable

variable += 1

variable = 1

function()

print(variable)
```

Note: removing the line containing the `global` keyword will spoil the code and the `UnboundLocalError` exception will be raised.

16. Changing the parameter's value **doesn't propagate it** outside the function. For example, the following snippet outputs `[1]` to the screen:

```
def function(parameter):

parameter = [2]

the_list = [1]

function(the_list)

print(the_list)
```

17. If the parameter is a **list** or a **dictionary**, changing its contents **propagates them** outside the function. For example, the following snippet outputs `[2]` to the screen:

```
def function(parameter):

parameter[0] = 2

the_list = [1]

function(the_list)

print(the_list)
```

18. **Recursion** is a programming technique in which the function **invokes itself** to perform a complex task. For example, the following snippet contains a function that evaluates the factorial of its argument and prints `120` to the screen:

```
def factorial(n):

if n < 2:

return n

else:

return n * factorial(n - 1)

print(factorial(5))
```

Exceptions and debugging

1. An **exception** is an event caused by an execution error which can induce program termination if not properly handled by the programmer. The situation in which the exception is created and propagated is called **raising** the exception.

2. Python professes its philosophy expressed with the sentence: *It's better to beg for forgiveness than ask for permission.* The recommendation hidden behind these words says that the programmer should **allow the exceptions to occur and handle them properly** instead of persistently avoiding problems and protecting the code from all possible errors with all their might.

3. To **control** exceptions and to handle them, Python provides a dedicated construction consisting of the `try` and `except` branches.

4. The `try` block encloses a part of the code that may cause an exception and when it happens, the execution of the block is terminated and the control jumps into the `except` block, which is dedicated to recognizing the problem and handling it. For example, the following snippet prints `PROCEEDING` even though the code provokes division by zero:

```
try:

x = 1 / 0

except:

x = None

print('PROCEEDING')
```

5. Here is a list of the most common Python exceptions:
   - `ZeroDivisionError`: raised by a division in which the divider is **zero** or is indistinguishable from zero (/, //, and %)
   - `ValueError`: raised by the use of values that are **inappropriate** in the current context, for example, when a function receives an argument of a proper type, but its value is **unacceptable**, for example, int('')
   - `TypeError`: raised by attempts to apply data of a **type** which cannot be accepted in the current context, for example, `int(None)`
   - `AttributeError`: raised – among other occasions – when the code tries to activate a **method** that doesn't exist in a certain item, for example, `the_list.apend()` (note the typo!)
   - `SyntaxError`: raised when the control reaches a line of code that **violates** Python's grammar, and which has remained undetected until now;
   - `NameError`: raised when the code attempts to make use of a **non-existent** (not previously defined) **item**, for example, a variable or a function.

6. When more than one exception is expected inside the `try` block and these different exceptions require different handling, another syntax is used where there is more than one named `except` branch. The unnamed (anonymous) `except` branch is the **default** one, and is responsible for servicing these exceptions which still need to be handled.

7. Not more than one `except` branch can be executed.

8. The default `except` branch – if it exists – must be the last branch.

9. For example, the following snippet outputs `NAN` when the user enters a string that is not an integer number, `ZERO` when the user enters `0`, and `ERR` in the case of another error:

```
try:

print(1 / int(input("Enter a number: ")))

except ValueError:

print('NAN')

except ZeroDivisionError:

print('ZERO')

except:

print('ERR')
```

10. An error existing in the code is commonly called a **bug**.

11. The process by which bugs are detected and removed from the code is called **debugging**.

12. The tool which allows the programmer to run the code in a fully controllable environment is called a **debugger**.

13. The '**print debugging**' technique is a trivial debugging technique in which the programmer adds some `print()` function invocations which help to trace execution paths and output the values of selected critical variables.

14. The process in which the code is probed to ensure that it will behave correctly in a production environment is called **testing**. The testing should prove that all execution paths have been executed and caused no errors.

15. The programming technique in which the tests and test data are created before the code is written or created in parallel with the code is called **unit testing**. It is assumed that every code amendment (even the most trivial) is followed by the execution of all previously defined tests.