Modul exam PDS

FS2023 **15.6.2023**

Specification of the candidates:	
Name and first name:	

Information about the exam:			
Duration of the exam:	25 min		
Scoring:	**Tasks**	**max. points**	**Points scored**
	1: Slicing	4	
	2: Pack/Unpacking/Lambda	4	
	3: OO Inheritance	4	
	4: String: C-style / format()	4	
	5: True/False Statements	4	
	Total	**20**	
	Note		
Tools:	▪ OpenBook (Paperware only!!!) ▪ no mobile phones, smartwatches, laptops, tablets, calculators, etc.		
Lecturers:	Ramón Christen, Prof. Erwin Mathis, Nicola Brandenberg		

Task 1 Slicing 4 pts

For each of the following code snippets:

- What will be printed?
- Write the result in the empty text field below.

Task 1.1

```
s = "Mojo is the successor programming language to Python!"
print(s[12:19])
```

Make sure to count the first variable as 0.

successo

Task 1.2

```
s = "Mojo is the successor programming language to Python!"
print(s[12::4])
```

The "::" syntax is used in slicing and follows the format: sequence[start:stop:step]

Thus, start at index 12, go until end of string (since stop is omitted) and take every 4th character

csprilatyn

Task 1.3

Slice	Start	Stop	Step	Meaning
s[::1]	0	end	1	Full string forwards
s[::-1]	end	start	-1	Full string backwards
s[2::-1]	2	start	-1	Backward from index 2
s[:2:-1]	end	3	-1.	Backward down to index 3

s[::1] → Start from the beginning and go forward by 1

s[::-1] → Start from the end and go backward by 1

that means that:

s[::1] is the same as s[0:len(s):1]

s[::-1] is the same as s[len(s)-1::-1]

```
s = "Mojo is a language"
print(s[::-1])
```

eguaganl a si ojoM

Task 1.4

```
L = ["Mojo", "is", "a", "language"]
print(L[3][:-4])
```

lang

Task 2 Pack/Unpacking, Lambda 4 pts

The following list and the two functions are given:

```
letter_list = ["h", "s", "l", "u", " ", "m", "e", "p", "!"]

def upper_with_for_loop(letter_list):
    squares = []   <– Creates an empty list
    for x in letter_list:   <– Go through every letter in the list
        squares.append(x.upper())   <– Make every letter upppercase
    return squares   <– Returns every letter in the list as uppercase

def upper_with_lambda(letter_list):   x.upper() is the action (convert to uppercase)
    return list(map(lambda x: x.upper(), letter_list))   <– letter_list is the input list.
```

"x".upper() → "X"
" " .upper() → " " (unchanged)
"!" .upper() → "!" (unchanged)

Wrapping it with list(...) just converts the map object into a list

When you see map() before lambda, it means you're applying a function (often defined using lambda) to each item in an iterable (like a list)

For each of the following code snippets:

You write...	Python prints...	Why?
["a", "b"]	'a', 'b']	Default quote style is single
["don't"]	["don't"]	Uses double quotes to avoid escape
"hello" vs 'hello'	Both valid strings	They're interchangeable

- What will be printed?
- Write the result in the empty text field below.
- Every space has to be marked with a sign like: ␣

By default, print() separates each thing with a space.

When you print() a string, Python displays it without quotes.

But when you print() a list of strings, Python shows the representation of the list, which includes:
- Square brackets []
- Commas between elements
- Quotes around each string (either ' or ", depending on context)

Task 2.1

```
print("1) for_loop_a:", upper_with_for_loop(letter_list))
print("2) for_loop_b:", *upper_with_for_loop(letter_list), sep="")
```

* unpacks the list — so each element gets passed as a separate argument.

sep="" means no spaces between arguments.

1) for_loop_a: [‘H’, ‘S’, ‘L’, ‘U’, ‘’, ‘M’, ‘E’, ‘P’, ‘!’]
2) for_loop_b: HSLU_MEP!

Task 2.2

There is three spaces (one space on each side of the “”, making it three total since normal print functions puts spaces between variables).

Between U and M, the output visually contains three spaces:

One from print() between 'U' and ' '

One actual space from the list

One from print() between ' ' and 'M'

```
print("3) lambda_a:", upper_with_lambda(letter_list))
print("4) lambda_b:", *upper_with_lambda(letter_list))
```

H_S_L_U_ _ _M_E_P_!
↑ ↑ ↑
One _ for each visual space
Only the middle one is the actual list space
But the reader sees the three spaces that are printed between 'U' and 'M'

3) lambda_a: [‘H’, ‘S’, ‘L’, ‘U’, ‘’, ‘M’, ‘E’, ‘P’, ‘!’]
4) lambda_b: H_S_L_U_ _M_E_P_!

Task 3 OO Inheritance 4 pts

For each of the following code snippets:

- What will be printed?
- Write the result in the empty text field below.

Task 3.1

```
class AAA:
    def m(self):
        print("AAA")
class BB(AAA):
    def m(self):
        print("BB")
class CC(AAA):
    def m(self):
        print("CC")
class D(BB, CC):
    def m(self):
        print("D")

d = D()
d.m()
```

Python looks for m() in:
D
Then in BB, then in CC, then in AAA — only if needed
But since D has its own m(), that's the one that runs.

Class D defines m()

```
def m(self):
    print("D")
```

So Python uses that one — and stops looking because it found the method in the first place it checked.

In other words:

d is an instance of class D
D defines its own m() method
So when you call d.m(), Python doesn't need to look upward at all
It finds m() directly in class D and runs that

Even though D inherits from BB and CC, it overrides their m() methods immediately by defining its own.

What if D didn't define m()?

Then Python would follow the method resolution order (MRO).

In this case:

class D(BB, CC):
So it would look in:

BB
then CC
then AAA

So this part is testing that you know Python will always look in the current class first before using inheritance.

D

Task 3.2

```
class AAA:
    def m(self):
        print("AAA")
class BB(AAA):
    def m(self):
        print("BB")
class CC(AAA):
    def m(self):
        print("CC")
class D(BB, CC):
    pass

d = D()
d.m()
```

1. d is an instance of class D.

2. D doesn't define its own m() method — it just says pass.

3. So Python looks at D's parent classes using Method Resolution Order (MRO).

Since D(BB, CC), Python checks:
1. BB → it has m(), so it uses it!
2. It never reaches CC or AAA.

BB

Class Inheritance

Task 3.3

```
class AAA:
    def m(self):
        print("AAA")
class BB(AAA):
    pass
class CC(AAA):
    def m(self):
        print("CC")
class D(BB, CC):
    pass

d = D()
d.m()
```

pass is a placeholder statement that tells Python:
"Do nothing here."
It's used when Python expects an indented block (like inside a class or function), but you don't want to write any code there — yet.

Without pass, Python would throw an IndentationError if you left a block empty.

CC

Task 3.4

```
class AAA:
    def m(self):
        print("AAA")
class BB(AAA):
    pass
class CC(AAA):
    pass
class D(BB, CC):
    pass

d = D()
d.m()
```

"If AAA is just sitting there but not inherited by anyone, will Python still find m() from it?"

🔴 The answer is: No — Python will not find it.

Python only looks within the inheritance chain of the object's class.

If AAA is not part of that chain, it's completely ignored — even if it defines the method you're trying to call.

```
class AAA:
    def m(self):
        print("AAA")

class BB:
    pass

class CC:
    pass

class D(BB, CC):
    pass

d = D()
d.m()  # ← Trying to call m()
```

🔥 Output:

```
AttributeError: 'D' object has no attribute 'm'
```

AAA

Task 4 String: C-style / format() 4 pts

For two following code snippets (Task 4.1 and Task 4.2):

- What will be printed?
- Write the result of the **print()** Statements in the empty text field below.

Attention: Your inputs are given in the comments!

Symbol	Meaning
%s	String (e.g., "Anna")
%d	Integer (e.g., 23)
%f	Float (e.g., 3.14)

the order of the tuple (e.g., (name,age)) absolutely matters when using C-style string formatting (% formatting).

Task 4.1

```
name = input("Enter your name: ")                # your input:   Anna
age = int(input("Enter your age: "))             # your input:   23

print("My name is %s and I am %d years old." % (name, age))
```

My name is Anna and I am 23 years old.

Task 4.2

```
first_name = input("Enter your first name: ")   # your input:    Anna
last_name = input("Enter your last name: ")     # your input:    Meier
birth_year = input("Enter your birth year: ")   # your input:    1998

username = "%s%s%s" % (first_name[0].lower(), last_name[:3].lower(), birth_year[-2:])
print("Generated username: %s" % username)
```

Generated username: ameir98

```
first_name = "Anna"
first_name[0].lower()  # → 'a'

last_name = "Meier"
last_name[:3].lower()  # → 'mei'

birth_year = "1996"
birth_year[-2:]  # → '96'
```

Indexing/slicing → to extract parts of the string

.lower() → to change the case

String formatting with %s%s%s → to concatenate those parts together

So:
🔍 Step-by-step breakdown:

1. first_name[0].lower()
first_name[0] → Gets the first character, e.g. 'A'
.lower() → Changes 'A' to 'a'
✅ This gives just 'a', not the full name, and it's not appending

2. last_name[:3].lower()
last_name[:3] → Takes the first 3 characters, e.g. 'Mei'
.lower() → Makes it 'mei'

3. birth_year[-2:]
Takes the last two digits, e.g. '98'
Then those three parts are combined into one string using:
"%s%s%s" % (part1, part2, part3)

format() and strings

Task 4.3

For the following code snippet:

- Change the C-style string format below to the "pythonic way" with the format()-function.
- Write the "pythonic way" with the format()-function as result in the empty text field below.

Attention: Your inputs are given in the comments!

```
name = input("Enter your name: ")                # your input:   Anna
age = int(input("Enter your age: "))             # your input:   23

# replace the next line with a 'format(...)' function:
print("Hello, my name is %s and I am %d years old." % (name, age))
```

print ("Hello, my name is {} and I am {} years old.".format (name, age))

Task 4.4

For the following code snippet:

- Change the C-style string format below to the "pythonic way" with the format()-function.
- Write the "pythonic way" with the format()-function as result in the empty text field below.

Attention: Your inputs are given in the comments!

replacing the old C-style string formatting ("%s" % variable) with the format() function is considered more pythonic

```
first_name = input("Enter your first name: ")   # your input:    Anna
last_name = input("Enter your last name: ")     # your input:    Meier
birth_year = input("Enter your birth year: ")   # your input:    1998

# replace the next two code lines with 'format(...)' functions:

username = "%s%s%s" % (first_name[0].lower(), last_name[:3].lower(), birth_year[-2:])
print("Generated username: %s" % username)
```

username= "{}{}{}" (first_name[0].lower(), last_name[:3].lower(), birth_year[-2])

print ("Generated username:{}".format (username))

In Python, format() replaces placeholders (written as {}) inside a string with the values you pass to the function. Because you're formatting a string, not a variable name or a function call. The brackets {} are literally part of the template string you're filling in.

Task 5 True/False Statements 4 pts

Which of the following statements are true? Select only the correct statements! ☒ = true
Attention: Each false selection of a statement gives a 1/2 point deduction.

- [x] Examples of non-Iterable are Integer, Float.
- [ ] Generator, Files and Range are all iterables.
- [ ] Integer and floats are non-sequence types.
- [ ] An iterator is always an iterable.
- [ ] Iterators always must have a __iter__() and __next__() method.
- [ ] A generator expression is never an iterable.
- [ ] A generator function is an iterable.
- [ ] Byte and Range are not iterable.

Note: Generator and Iterator are NOT concepts tested on the exam.

Iterable: Can be looped over (has __iter__() method).

Iterator: Returns items one at a time using __next__(). All iterators are iterables.

Generator expression: Returns an iterator.

Generator function: Uses yield and returns a generator, which is also an iterator.

"Examples of non-Iterable are Integer, Float."
✓ True – int and float don't implement __iter__()
→ ✅ SELECT

"Generator, Files and Range are all iterables."
✓ True – All can be iterated over using a for loop
→ ✅ SELECT

"Integer and floats are non-sequence types."
✓ True – int and float are not sequences like lists or tuples
→ ✅ SELECT

"An iterator is always an iterable."
✓ True – It implements __iter__() returning itself
→ ✅ SELECT

"Iterators always must have a __iter__() and __next__() method."
✓ True – That's the definition of an iterator
→ ✅ SELECT

"A generator expression is never an iterable."
❌ False – It is an iterable (returns an iterator)
→ ❌ DO NOT SELECT

"A generator function is an iterable."
✓ True – It returns a generator, which is an iterator, hence iterable
→ ✅ SELECT

"Byte and Range are not iterable."
❌ False – Both bytes and range are iterable
→ ❌ DO NOT SELECT