An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in Python

Guilherme Gama *Ricardo Caceffo* *Renan Souza*
Raysa Bennati *Tales Aparecida* *Islene Garcia*
Rodolfo Azevedo

Technical Report - IC-18-19 - Relatório Técnico
November - 2018 - Novembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.
O conteúdo deste relatório é de única responsabilidade dos autores.

An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in Python

Guilherme Gama, Ricardo Caceffo, Renan Souza, Raysa Benatti,

Tales Aparecida, Islene Garcia, Rodolfo Azevedo

Institute of Computing – State University of Campinas (UNICAMP),
Caixa Postal 6176, 13083-970 Campinas-SP, Brasil

guilheremegama@gmail.com; caceffo@ic.unicamp.br; renan-plsouza@uol.com.br; raysa.benatti@students.ic.unicamp.br; tales.aparecida@gmail.com; islene@ic.unicamp.br; rodolfo@ic.unicamp.br

Abstract. The purpose of this work is to document the development process of a Concept Inventory for introductory Computer Science in the Python programming language. It covers the analysis of our previous work in the C language, our inspection of past exam answers from two sections of a CS1 course in Python, and interviews with the instructors who gave these sections. From our previous list of 33 misconceptions in C, we have maintained 17, and raised new 11 hypotheses specific to the Python language, for a total of 28 hypothetical misconceptions to be validated as a further step of this study.

Keywords: Misconception, Introductory Programming, Concept Inventory, Antipattern

1. Introduction

The purpose of this report is to document the development and assessment of a Concept Inventory (CI) for CS1 Introductory Programming Courses in the Python programming language, as part of our larger research effort towards the development of a CI for CS1 Introductory Programming Courses in a more general sense [1,2,3,4], carried out under the supervision of Rodolfo Azevedo and Ricardo Caceffo, from the Computing Institute of the University of Campinas (UNICAMP). A Concept Inventory is a set of multiple-choice questions addressing specific misunderstandings and misconceptions of the students [5,6,7].

The investigative and analytical steps undertaken regarding the Python language include (a) an assessment of whether any part of our previous work in the C language would be applicable to Python, (b) an analysis of exam answers from previous sections of MC102 given in Python, (c) interviews with the instructors who were responsible for those sections and ceded the exams mentioned in (b), and (d) a compilation of the results from (a), (b), and (c) into a final list of hypothetical misconceptions in Python. Figure 1 illustrates this process:

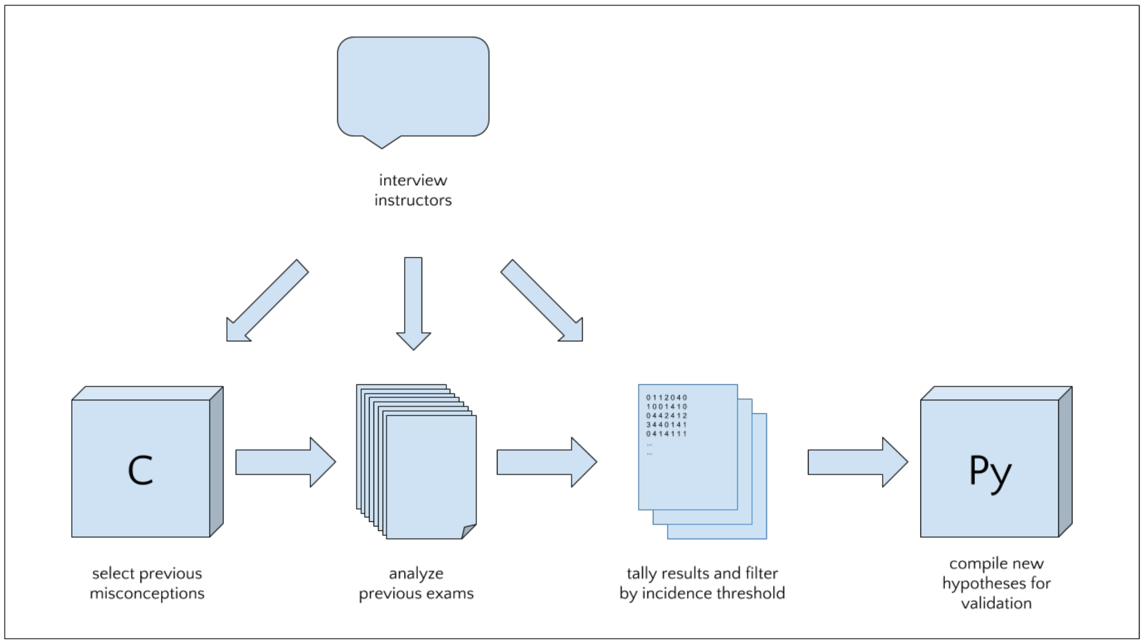

interview instructors
C
select previous misconceptions
analyze previous exams
tally results and filter by incidence threshold
Py
compile new hypotheses for validation

Figure 1: Methodology used in raising new hypothetical misconceptions in Python

As a result of this study, 17 misconceptions from C were carried over to Python, 16 were discarded, and 11 hypotheses for new misconceptions in Python were raised. As the focus of this research project is on the challenges faced by undergraduate students when taking their first steps in Computer Science, that the only path to verifiable data in that respect is going to the persons involved in that process—namely, the students and instructors themselves. Hence, the resulting list of 28 possible misconceptions is now pending validation in a real-life undergraduate environment so as to be consolidated into a Concept Inventory in their own right.

2. Methodology

The starting point for this stage of the process was the already existing Concept Inventory in C [2], containing 33 documented and validated misconceptions, organized in 7 categories, in that programming language. These misconceptions were individually considered in terms of their applicability to the Python language from a technical point of view (*e.g.* direct handling of pointers is permitted in C but not in Python).

In sequence, 204 exams from previous sections of MC102 in Python were individually inspected to ascertain whether (a) which misconceptions in C might be applicable to Python, (b) which misconceptions were not, and (c) whether there were any new potential misconceptions in Python that had not been detected in C.

The incidence of these hypothetical misconceptions—or lack thereof—was quantified in absolute numbers, per student per exam (but disregarding multiple incidents by a student in the same exam). The resulting data was then filtered in terms of a percentage compared to the total number of exams analyzed, and a final list of hypothetical misconceptions was compiled. The steps taken in this analytical process are described in full in Section 3.

The three instructors who had taught MC102 in Python were interviewed regarding their experiences with respect to those sections of the course. English translations of the full transcripts of those interviews are given in Section 4. The goal of these interviews was to provide insights into the learning process of CS1 students in Python to assist in the decision process regarding the development of this CI.

Finally, each hypothetical misconceptions that was maintained for validation was described in full, following the Antipattern [5] method, following the same template as in our work in the C language [2].

3. Exam Answer Analysis

We individually analyzed all students' answers to five exams of the introductory CS1 course MC102 (Algorithms and Computer Programming), encompassing two sections in which that course was offered in Python. Copies of the original exam papers are in Appendix A and their transcriptions in English, in Appendix B.

The objective of this analysis was to answer two questions: Q1) Were any of the C-language misconceptions from our previous work applicable to Python, and Q2) Are there any Python-specific misconceptions that had not been detected in C?

Summaries of the relevant data collected from the MC102 exams in Python are given in Subsections 3.1 (relevant to I1's section in the 2nd semester of 2015) and 3.2 (I2's section in the 2nd semester of 2016). The process of interpreting those data and consolidating a final list of hypothetical misconceptions for validation is outlined in Subsection 3.3.

To answer questions Q1 and Q2, we applied the following analysis methodology: for each answer to each question of each exam, we attributed one or more codes depending on these cases:

- **C**, if the student was awarded full marks for that question;
- **N**, if the question was left blank or if there was no good-faith attempt to answer the question;
- The code of a misconception in C language from our corresponding Concept Inventory (as listed in Table 1) if such an instance were detected;
- A code in the form **X.i**, $i = 1, 2, 3$ ..., if a new possible misconception in Python were raised;
- And **O**, in any other case.

Table 1: Misconceptions in C Language from the Concept Inventory for Computer Science, not including categories E and F, which were not used.

Category	Code	Description
A: Function Parameter Use and Scope	A.1	Parameter value set by external source
	A.2	Parameters passed as if by reference
	A.3	Attempt to access parameter from outside scope
	A.4	Incorrect order of function parameters
	A.5	Function return value not handled by caller function
	A.6	Logic error related to parameters when calling function
B: Variables, Identifiers, and Scope	B.1	Attempt to access local variables from outside scope
	B.2	Global variables considered local in current scope
	B.3	Parameter mistaken for same-name variable outside function
	B.4	Global variables assumed inaccessible from within function
C: Recursion	C.1	Wrong expression used to calculate the return value of a recursive function
	C.2	No recursive call
	C.3	No termination at base case
D: Iteration	D.1	Improper update of loop counter
	D.2	Use of loop result before completion
	D.3	Loop iterates the correct number of times, but over the wrong range.
	D.4	Loop iterates an incorrect number of times.
	D.5	Absence of loop (single iteration)
	D.6	Loop construction incoherent with remainder of code
G: Boolean Expressions	G.1	Incorrect precedence for Boolean operators
	G.2	Nested if-statements instead of Boolean
	G.3	Arithmetic expression instead of Boolean
	G.4	Attempt to evaluate Boolean through loops

In the event that more than one misconception were detected, multiple codes were assigned as necessary. It should be pointed out that the **O** code encompasses any range of mistakes that were not attributable to known misconceptions or new hypotheses thereof.

These codes were registered in exams analysis tables—one for each exam session—where each column corresponded to a question of that exam, and each row to a student who took that exam. Table 2 shows an example of this record. The complete tables are provided in Appendix C.

Table 2: Sample exam analysis table

Anonymized Student Identifier	Question			
1	C	C	O	A2
2	C	O	C2	X.1
3	O	O	D2 D3	X.2

Any new possibility of a misconception was registered in a separate table with its attributed code, a short description, and an example in Python, as shown on Table 3:

Table 3: Sample record of new misconception

Code	Description	Example

X.1	Attempt to use an instance method outside of its class.	`t = (2, 3)`
		`t.append(4) # AttributeError`

The results of this process are summarized in the following subsections. For each exam, we present a summary of the statistics generated for each exam analyzed. The new misconception codes **X.1...X.17** are summarized in Table 9.

3.1 MC102, 2nd Term 2015, I1

Instructor I1 ceded a total of 104 student exam papers relative to the three exams offered during his section of MC102 in Python, in the second term of 2015.

3.1.1 Midterm Exam

51 students took the midterm exam of I1's 2nd Term 2015 section of MC102. Their responses and the hypothetical misconceptions found are summarized in Table 4.

Table 4: Summary: Midterm Exam, MC102 2/2015, I1

Question	**Response rate**	**Existing C-language misconceptions**	**Hypothesized Python misconceptions**
1	Answered: 51 (100%)	A.1: 1	X.1: 21

	Correct: 2 (3.92%)	D.1: 4 D.2: 15 D.3: 9 D.4: 5 D.5: 1 D.6: 2	
2	Answered: 49 (96.08%) Correct: 10 (19.61%)	A.1: 1 D.1: 4 D.2: 15 D.3: 9 D.4: 5 D.5: 1 D.6: 2	X.2: 1
3	Answered: 49 (96.08%) Correct: 12 (23.53%)	A.5: 1 D.2: 7 D.3: 5 D.4: 2 D.5: 3 D.6: 1	X.3: 1
4	Answered: 48 (94.12%) Correct: 16 (31.37%)	A.5: 2 D.2: 2 D.3: 1	X.2: 1 X.3: 3

3.1.2 Final Exam

A total of 41 students took the final exam of I1's 2nd Term 2015 section of MC102. Their responses and the hypothetical misconceptions found are summarized in Table 5.

Table 5: Summary: Final Exam, MC102 2/2015, I1

Question	Response rate	Existing C-language misconceptions	Hypothesized Python misconceptions
1	Answered: 39 (95.12%) Correct: 15 (36.59%)	A.4: 1 D.3: 2	X.2: 1 X.4: 3 X.5: 5 X.6: 2 X.7: 2 X.8: 1
2	Answered: 39 (95.12%) Correct: 15 (36.59%)	C.1: 2 C.2: 8 C.3: 2 D.3: 2	X.2: 2 X.3: 1
3	Answered: 34 (82.93%) Correct: 3 (7.32%)	D.2: 1 D.3: 2 D.4: 3 G.2: 1	X.6: 1
4	Answered: 39 (95.12%) Correct: 7 (17.07%)	D.1: 2 D.2: 1 D.3: 5 D.4: 1 D.5: 3 D.6: 1	X.3: 1 X.4: 4

3.1.3 Retake Exam

A total of 12 students took the retake exam of I1's 2nd Term 2015 section of MC102. Their responses and the hypothetical misconceptions found are summarized in Table 6.

Table 6: Summary: Retake, MC102 2/2015, I1

Question	Response rate	Existing C-language misconceptions	Hypothesized Python misconceptions
1	Answered: 12 (100%) Correct: 3 (25%)	None detected.	X.1: 3
2	Answered: 12 (100%) Correct: 9 (75%)	None detected.	None detected.
3	Answered: 12 (100%) Correct: 7 (58.33%)	D.4: 1	None detected.
4	Answered: 12 (100%) Correct: 5 (41.67%)	None detected.	X.2: 1 X.4: 2

3.2 MC102, 2nd term 2016, I2

Instructor I2 ceded 100 student exam papers relative to the midterm and final exams.

3.2.1 Midterm Exam

A total of 53 students took the midterm exam of I2's 2nd Term 2016 section of MC102. Their responses and the hypothetical misconceptions found are summarized in Table 7.

.

Table 7: Summary: Midterm, MC102 2/2016, I2

Question	Response rate	Existing C-language misconceptions	Hypothesized Python misconceptions
1	Answered: 50 (94.34%) Correct: 17 (32.08%)	None detected.	None detected.
2	Answered: 53 (100%) Correct: 14 (26.42%)	D.1: 3 D.2: 4 D.4: 6 D.5: 3 D.6: 2 G.1: 1	X.1: 1 X.4: 2 X.6: 1 X.9: 2 X.10: 1
3	Answered: 45 (84.91%) Correct: 7 (13.21%)	D.1: 3 D.2: 2 D.3: 4 D.4: 12 D.6: 4	X.1: 4 X.4: 1 X.5: 1 X.9: 1 X.11: 1 X.12: 1 X.13: 2
4	Answered: 53 (100%) Correct: 31 (58.49%)	None detected.	None detected.

3.2.2 Final Exam

A total of 47 students took the final exam of I2's 2nd Term 2016 section of MC102. Their responses and the hypothetical misconceptions found are summarized in Table 8.

.

Table 8: Summary: Final, MC102 2/2016 I2

Question	Response rate	Existing C-language misconceptions	Hypothesized Python misconceptions
1	Answered: 47 (100%) Correct: 36 (76.6%)	None detected.	None detected.
2	Answered: 43 (91.49%) Correct: 17 (36.17%)	A.1: 1 D.4: 1 D.5: 1	None detected.
3	Answered: 45 (95.74%) Correct: 21 (44.68%)	A.5: 1 C.1: 2 C.2: 2 C.3: 1 D.4: 1	None detected.
4	Answered: 45 (95.74%) Correct: 13 (27.66%)	D.2: 17	X.1: 9 X.7: 1 X.14: 1 X.15: 5 X.16: 1 X.17: 2

3.2.3 Retake Exam

No students sat the retake exam in Python for that section.

3.3 Selection of Hypotheses for Validation

In total, we raised 17 new possible misconceptions in this stage of analysis, which had not been detected in our previous work in the C programming language. These are listed in Table 9:

Table 9: List of new misconception hypotheses

Code	Description	Example

X.1	The `for` statement is considered a function, and the iterating variable is considered to be local to the scope of that supposed function.	`x = 5`
		
		`for x in range(10):`
		`    (...)`
		
		`# x is expected to be 5 here`

X.2	`for` statement is called for a `range` object, but loop code is constructed as if it were defined over the members of an `iterable`, such as a list.	`for i in range(len(my_list)):`
		`    if i > 5:`
		`        print(i)`

X.3	Using the `def` keyword to call an already defined function.	`# swap function defined earlier`
		
		`a = 3, b = 5`
		`def swap(a,b)`

X.4	Attempting to invoke a method from an instance of class in which it is not defined.	`a = (3,4,5)`
		`a.append(6) # AttributeError`

X.5	Calling an instance method that returns a new instance of that class and not attributing that instance to a variable.	`s = “Hello world”`
		`s.replace("H", "h") # s remains unchanged`

X.6	`for` statement is called to iterate over members of an `iterable`, such as a list, but loop code is constructed as if were defined over a `range` object.	`for i in my_list:`
		`    print(my_list[i]) # possible IndexError`

X.7	Method is called without having been imported, or, without referencing a module with dot notation if the module itself was imported.	`import math`
		
		`a = sqrt(3) # NameError`

X.8	Attempt to alter `str` object by index	`s = “Hello world”`
		`s[0] = “h” # TypeError`

X.9	Attempt to attribute a value to a method, rather than pass that value as a parameter.	`a = [3,4,5]`
		`a.append = 6 # AttributeError`

X.10	Incorrect use of slice notation.	`a = [3,4,5]`
		`print(“The first two elements of a are “, a[2:])`

X.11	Calling the `for` statement and not providing an iterable object.	`for i in 10: # TypeError`
		`    (...)`

X.12	Attempting to concatenate `str` objects without the + operator.	`first_name = “Jane”`
		`last_name = “Doe”`
		`full_name = last_name “, “ first_name`

Code	Description	Example

X.13	Acting on the assumption that a previously assigned return value of a function will be automatically updated when the parameters given to the function are changed.	`def sum_of_squares(a, b):`
		`    return a**2 + b**2`
		
		`a = 3`
		`b = 4`
		`c = sum_of_squares(a, b)`
		`a = 6`
		`b = 8`
		`print(“The result of 6**2 + 8**2 is “, c)`

X.14	Comparing two referenced objects by their references when comparison methods have not been defined for that class.	`# Dog class defined elsewhere`
		
		`a = Dog(name = “Fido”)`
		`b = Dog(name = “Fido”)`
		`if a == b: # returns False`
		`    (...)`

X.15	Not using the `self` keyword when referencing instance attribute within class definition.	`class Dog:`
		`    name = “Default Dog”`
		
		`    def __str__(self):`
		`        print(name) # NameError`

X.16	Not indicating `self` as the first argument of an instance method.	`class Dog:`
		`    (...)`
		`    def get_name(): # TypeError when called`
		`        return self.name`

X.17	Attempt to reference instance attribute by subscript when no `__getitem__` method has been defined.	`class Dog:`
		`    name = “Default Dog”`
		`    (...)`
		
		`dog = Dog()`
		`print(“The name of this dog is “, dog[0]) # TypeError`

We tallied our final count of misconceptions from Table 1 and Table 9. These counts reflect occurrences per student per exam, inasmuch as multiple occurrences by one student in a single exam were counted as a single event for the purposes of this final tally. Both the C and Python misconceptions were now considered hypothetical, and were only maintained as candidates for validation if the number of occurrences reached a threshold of 1% of all exam papers, namely, a count greater than or equal to 2 (out of 204). The results of that tally are listed in Table 10.

Table 10: Tally of misconception hypotheses and their status regarding further validation

Misconception code	Number of occurrences	Final status
A.1	2	Maintained

A.2	0	Rejected
A.3	2	Maintained
A.4	1	Rejected
A.5	3	Maintained
A.6	0	Rejected
B.1	5	Maintained
B.2	0	Rejected
B.3	2	Maintained
B.4	0	Rejected
C.1	4	Maintained
C.2	10	Maintained
C.3	3	Maintained
D.1	12	Maintained
D.2	41	Maintained
D.3	28	Maintained
D.4	32	Maintained
D.5	11	Maintained
D.6	10	Maintained
G.1	1	*Maintained despite not reaching threshold*
G.2	1	
G.3	0	
G.4	0	
X.1	38	Maintained
X.2	6	Maintained
X.3	6	Maintained
X.4	10	Maintained
X.5	6	Maintained
X.6	4	Maintained
X.7	3	Maintained
X.8	1	Rejected
X.9	3	Maintained
X.10	1	Rejected
X.11	1	Rejected
X.12	1	Rejected
X.13	2	Maintained
X.14	1	Rejected
X.15	5	Maintained
X.16	1	Rejected
X.17	2	Maintained

The results shown on Table 10 informed our decision as follows:

3.3.1 Selection of C language misconceptions for validation by category

3.3.1.1 Category A: Function Parameter Use and Scope

C language misconceptions A.1, A.3, A.5 were renamed as Python misconception hypotheses PA.1, PA.3, PA.5 and maintained for further validation. A.2, A.4, A.6 were excluded from further consideration for not reaching the threshold.

3.3.1.2 Category B: Variables, Identifiers, and Scope

C language misconceptions B.1 and B.3 were renamed as Python misconception hypotheses PB.1 and PB.3. B.2 and B.4 were excluded from further consideration.

3.3.1.3 Category C: Recursion

C language misconceptions C.1, C.2, C.3 were renamed as Python misconception hypotheses PC.1, PC.2, and PC.3.

3.3.1.4 Category D: Iteration

C language misconceptions D.1, D.2, D.3, D.4, D.5, D.6 were renamed as Python misconception hypotheses PD.1, PD.2, PD.3, PD.4, PD.5, and PD.6.

3.3.1.5 Category E: Structures

This category was not included in any step of this process, due to its irrelevance in the Python language.

3.3.1.6 Category F: Pointers

This category was not included in any step of this process, due to its irrelevance in the Python language.

3.3.1.7 Category G: Boolean Expressions

This was perhaps the most surprising result. As per our threshold criterion, the entirety of Category G would have been excluded from further consideration. Notwithstanding, we opted to defer that decision until the forthcoming validation of these misconceptions, where we expect either to obtain more solid evidence to justify either the removal of the category from the Python CI as a whole or in part, or its maintenance. Misconceptions G.1, G.2, G.3, G.4 were thus renamed PG.1, PG.2, PG.3, PG.4 and kept for validation.

3.3.2 Selection of new hypothetical misconceptions

3.3.2.1 Exclusions

Misconception hypotheses X.8, X.10, X.11, X.12, X.14, X.16 were excluded from further consideration.

3.3.2.2 Hypotheses assigned to existing categories from the C language CI

Hypotheses X.13 and X.1 were added to Category B, renamed as PB.4 and PB.5, respectively.

Hypotheses X.6 and X.2 were added to Category D, renamed as PD.7 and PD.8, respectively.

3.3.2.3 Hypotheses assigned to new categories

The remaining hypotheses fall under issues related to the use of Python modules, classes, and attributes, or to the definition of new ones. Hence, we created a new category H: Use and Implementation of Classes and Objects, to which we added hypotheses X.4, X.5, X.3, X.7, X.9, X.15, X.17 as PH.1, PH.2, PH.3, PH.4, PH.5, PH.6, PH.7, in that order.

3.3.3 Result

The application of the selection decisions and renaming process outlined above, we reached a final list of hypothetical misconceptions in Python as shown in Table 11.

.

Table 11: Final list of hypothetical misconceptions in Python for validation and comparison with original C language codes when applicable. Legend: (*) Insufficient evidence to maintain this hypothesis for Python language, (**) Maintained for validation despite lack of evidence thereof, (***) hypothesis raised during exam analysis.

Topic	**Misconception**	**C Language**	**Python Language**
A) Function Parameter Use and Scope	Parameter value set by external source	A.1	PA.1
	Parameters passed as if by reference	A.2	*
	Parameters accessible outside their scope	A.3	PA.3
	Incorrect order of function parameters	A.4	*
	No catching of function return value	A.5	PA.5
	Logic error related to parameters when calling a function	A.6	*
B) Variables, Identifiers, and Scope	Attempt to access local variables from outside scope	B.1	PB.1
	Global variables considered local in current scope	B.2	*
	Confusing parameter with same-name variable outside function	B.3	PB.3
	Global variables considered inaccessible from within function	B.4	*
	For statement iteration variable considered local	Not applicable	PB.5 ***
	Variable that has received the return value of a function will automatically update when the variables passed as arguments to that function are updated.	Not applicable	PB.6 ***
C) Recursion	Wrong expression used to calculate return value of recursive function	C.1	PC.1

Topic	Misconception	C Language	Python Language
	No recursive call	C.2	PC.2
	No base case termination	C.3	PC.3
D) Iteration	Improper loop counter update	D.1	PD.1
	Premature use of loop result	D.2	PD.2
	Loop iterates correct number of times, but over wrong range.	D.3	PD.3
	Loop iterates incorrect number of times.	D.4	PD.4
	Absence of loop (single iteration)	D.5	PD.5
	Loop construction incoherent with remainder of code	D.6	PD.6
	For loop iterating over members of an iterable object is treated as a for loop over a range() instance.	Not applicable	PD.7 ***
	For loop iterating over range() instance is treated as a for loop over members of an iterable object.	Not applicable	PD.8 ***
E) Structures	Structs compared by identifier	E.1	Not applicable
	Struct identifier compared to field	E.2	
	Struct accessed as pointer	E.3	
	Struct field accessed as array index	E.4	
	Use of "struct" keyword after declaration	E.5	
F) Pointers	Use of "&" to dereference	F.1	

Topic	**Misconception**	**C Language**	**Python Language**
	No dereferencing	F.2	Not applicable
	Invalid address assigned to pointer	F.3	
	Void function returns value	F.4	
	Parameters passed as if reference	F.5	
G) Boolean Expressions	Incorrect precedence for Boolean operators	G.1	PG.1 **
	Nested if-statements instead of Boolean expression	G.2	PG.2 **
	Arithmetic expression instead of Boolean	G.3	*
	Attempt to evaluate Boolean through loop	G.4	PG.4 **
H) Use and Implementation of Classes and Objects	Attempt to invoke method outside its class	Not applicable	PH.1
	Treating method that returns a new instance of the same object as one that changes the instance itself.		PH.2
	Function call preceded by `def` keyword		PH.3
	Class attribute invoked without being imported or with no class specified.		PH.4
	Assignment of value to method		PH.5
	No `self` keyword to reference instance attributes		PH.6
	Attempt to call class attribute through indices		PH.7

The two chief questions that we proposed in the beginning of this section can now be further developed:

Q1) "Were any of the C-language misconceptions from our previous work applicable to Python?"

The maintenance of Categories A (functions), B (variables and scope), as shown in Table 11, seems to point firmly in the direction of an affirmative answer to Q1. Categories A and B were maintained in part, given the fact that the effective use of Python language involves a reasonable degree of knowledge of functions, variables, and scope, although the handling of these features holds some pragmatic differences with how it is done in C. One notable distinction, for instance, is the absence of a mandatory `main()` function in Python, and the ramifications it creates in C, in terms of the handling of variable scope. The absence of misconception B.4 in the potential Python CI is a direct consequence of that distinction: Python students begin programming at the global scope, restricting the scope only when defining functions or loops, whereas C students start at the local scope of `main()` and are customarily only permitted to declare and use global variables much later.

It is in Categories C (recursion) and D (iteration), however, that we find our best evidence for Q1. Not only have the previous misconceptions in those categories been maintained, but they have done so with remarkable consistency. Category D in particular, has some of the most frequently occurring items, most notably D.2, having been found in 41 (20%) of all students' responses. This is, of course, not entirely surprising—given the general nature of recursion and iteration in computer programming, it would be expected that these subjects would pose challenges of the same nature in the students' learning process. This notion was also unanimously corroborated by the three instructors interviewed in Section 4.

Our most surprising finding in this regard was when it came to category G (Boolean expressions). Not one of the misconceptions survived our threshold of acceptance for validation. It would not have been unreasonable to expect that, due to the very general nature of Boolean expressions in Computer Science, students' difficulties in C and in Python would have some degree of correlation, but our findings give us no evidence to support that assumption. We have two hypotheses as to why this may have been the case. Firstly, with the exception of Question 3 in the MC102 2nd Term 2015 Final Exam, none of the exam questions required the use of complex Boolean expressions that would create the opportunity for these misconceptions to manifest themselves. Secondly, it may be that the nature of Python syntax (the use of the English words `and`, `or`, `not` in contrast with their symbolic counterparts in C) makes it easier for students to correctly assess the correctness of their work and make the use of longer expressions with parentheses less daunting. Either of these hypotheses is entirely speculative, of course, and can only be truly tested with further investigation in our validation stage.

We move on to Q2) "Are there any Python-specific misconceptions that had not been detected in C?"

Once again we turn to Table 11. Category B sees the inclusion of two new possible misconceptions. B.5 is strictly related to the handling of iteration in Python, as *ad hoc* iteration variables in statements such as `for i in range(x)` are local to the scope where the `for` instruction was invoked, rather than internal to the scope of the loop. This will not be an issue for students learning C, as the counting variable will have either been declared prior to the loop statement, or, if implicitly created as in `for (int i = 0; …)`, will indeed be visible only inside the loop.

Category D shows two new possible misconceptions, PD.7 and PD.8, occurring with reasonable frequency (both around 5%), which would be directly related to the specifics of the Python syntax. In short, any explicit `for` loop[1] in Python will iterate over the elements of an object implementing the `__iter__` method—such as members of a list. Iterating over a variable counting over a sequence of integers requires the use of the built-in class range, which provides the programmer with a sequence such as *0, 1, 2, 3, ..., n-1*. The distinction between these two uses of the `for` statement, or rather, these two variations on the *same* use of the for statement, proved challenging for some exam-takers, either in the form of calling `for i in some_list` and then attempting to access `some_list[i]` (PD.7) or declaring for

[1] The use of list comprehensions, one of the signature features of Python, was not, to our knowledge, taught in any of the MC102 sections we examined, and has therefore not been brought into consideration in this study. It is necessary to point this out especially in the context of hypothetical misconception PB.5, as the variable `i` in a statement such as `[i for i in range(n)]` would indeed be local to the scope of the loop.

`i in range(len(some_list))` and then attempting to use `i` as if directly referencing the element itself (PD.8).

Finally, Q2 is most strongly addressed in the form of Category H (use and implementation of classes and objects). This is perhaps where the specifics of the Python language become more relevant in the students' learning experience. Misconception hypotheses PH.1, PH.2, PH.4, and PH.5 all deal with issues related to the definition of how Python structures its classes and attributes. It is not unreasonable that a student might want to call the `append()` method from a tuple—after all, it's also a collection of elements—nor is it unexpected that they would call `replace()` on a string and not attribute the result to a new identifier if they have been modifying lists through the exact same dot syntax for the past few weeks. Misconceptions PH.6 and PH.7 deal with the explicit construction of user-defined classes and the specific demands of the Python language. The parallel between PH.7 and E.4 is particularly noteworthy, but have been kept separate at this time due to the very distinct nature of the C language `structs` and Python objects. Finally, misconception PH.3 (the incorrect use of the def keyword when calling an existing function) is a direct consequence of the dynamically-typed nature of Python—statically-typed languages such as C or Java require that the programmer specify the return type of a function as the beginning of its declaration.

All of these considerations seem to indicate that the Q2 can be answered in very broad terms as, "Yes, there are new misconceptions that are specific to the Python language". As with Q1, our findings here still require a final step of validation with the students themselves.

4. Interviews with Course Instructors

We interviewed instructors I1, I2, I3 of the UNICAMP Institute of Computing, all of whom had taught at least one section of MC102 in Python in previous years. These interviews were semi-structured, intended to cover all aspects of our existing CI material, yet allowing for new insights on the pedagogical aspects of teaching introductory CS in Python. Subsections 4.1, 4.2, and 4.3 contain the English translations for the full transcripts for instructors I1, I2, and I3, respectively.

4.1 I1

Date: 19 July 2017

1. **Introductory notes**
   Here, Ricardo [Caceffo] explains the relevant details of the project and the reason for this interview.

2. ***May we record this interview?***
   The instructor conceded to the recording.

3. ***Have you taught MC102 or other introductory-level CS courses in other programming languages besides Python? If so, which?***
   The question was not asked, but the instructor did mention having taught MC102 in C.

4. ***We've separated CS1 topics in 7 categories, having identified 33 misconceptions altogether. These categories are (A) function parameter use and scope (B) Variables, identifiers, and scope (C) Recursion (D) Iteration (E) Structures (F) Pointers (G) Boolean expressions.***

   4.1. ***Regarding Category A (Function parameter use and scope), which misconceptions did you find your students had? Do you believe these are the same as we've found in C? To what point are we able to generalize (find common ground between C and Python)?***
   The instructor stated that in Python, arguments are always passed by reference, and this could, in theory, cause some doubts for the students. He did not, however, recall any specific issues his students might have had.

   4.2. **Question 4.1 repeated for category B**
   The instructor mentioned issues related to indentation in Python, which is the most important factor in determining the scope of a piece of code (as is done by curly braces in C). He said this created some confusion, especially in the handwritten exams.

   4.3. **Category C**
   In the instructor's opinion, problems with recursion have a more general nature and have no specific issues in Python.

4.4. **Category D**

The instructor mentioned he encouraged students to create loops explicitly when necessary, but in general preferred that they used structures that would iterate over members of an `iterable` object. He mentioned there was some confusion regarding the use of the `range()` constructor, and many students would make small adjustments in the counting variable in the body of the loop (*e.g.,* incrementing or decrementing it by 1), in order to get their codes past the automatic correction.

4.5. **Category E**

According to the instructor, complex types such as objects were briefly mentioned at the end of the term. He doesn't remember if they wrote any constructor methods, nor if there was any difficulty regarding the use of the `self` keyword from a conceptual point of view, even though students would frequently forget to use it when needed.

4.6. **Category F**

Although Python does not allow for the direct manipulation of pointers, the instructor mentioned that, in the beginning of the term, the `id()` function was used so that students could understand how Python stores variables.

4.7. **Category G**

The instructor confirmed that there had been students who would substitute Boolean expressions for large chains of if-else statements, often resulting in needlessly long code. He did not recall if there were doubts regarding operator precedence and / or use of parentheses.

5. ***On a final note, one possibility we've considered would be to create "analogy questions" which are independent of the language use. We've done some experiments in that regard, but we've come to the conclusion, that solving these questions depends more on correctly interpreting the prompt and on the students' understanding of logic, than effectively their grasp on programming concepts. Would you have any suggestions?***

The instructor mentioned there are conceptual problems common to all programming languages that can be considered. He mentioned the case of recursion and told us about his own personal experience with the subject.

4.2 I2

Date: 20 July 2017

1. **Introductory notes**
   Here, Ricardo [Caceffo] explains the relevant details of the project and the reason for this interview.

2. ***May we record this interview?***
   The instructor conceded to the recording.

3. ***Have you taught MC102 or other introductory-level CS courses in other programming languages besides Python? If so, which?***
   I2 taught MC102 in C language in the second term of 2017, having also taught the course in Python in the second term of 2016.

4. ***We've separated CS1 topics in 7 categories, having identified 33 misconceptions altogether. These categories are (A) function parameter use and scope (B) Variables, identifiers, and scope (C) Recursion (D) Iteration (E) Structures (F) Pointers (G) Boolean expressions.***

   4.1. ***Regarding Category A (Function parameter use and scope), which misconceptions did you find your students had? Do you believe these are the same as we've found in C? To what point are we able to generalize (find common ground between C and Python)?***
   The instructor mentioned the Python keyword `global` but could not recall whether that had been taught in the course. He said passing arguments by reference might create difficulties for students (such as passing a list as an argument, for instance).

   4.2. ***Question 4.1 repeated for category B***
   He mentioned that, on the one hand, Python made things easier by not requiring a variable to be declared before assignment, but, on the other hand, the fact that Python is dynamically-typed might create difficulties. He did not specify whether this had been the case when he himself taught the course.

   4.3. Category C
   According to him, there were no differences concerning that category with regard to teaching MC102 in C language.

   4.4. Category D
   He mentioned the absence of a do-while loop structure in Python, which, if available, would have made students' lives easier, but he mentioned there were difficulties in using the overloaded `range()` constructor, which can receive from 1 to 3 arguments. `range()`, according to the instructor, "masks" the loop initialization parameters, and students don't always clearly realize what they are.

4.5. Category E
The instructor mentioned that objects were used as a teaching tool, to enable students to understand complex types such as `struct` in C language. They were not taught formally as objects in the context of Object-Oriented Programming. Hence, for instance, students learned standard procedures to create a deep copy of a matrix, without understanding the difference between a deep copy of an object and simplify attributing its reference to another variable.

4.6. Category F
Pointers were not taught.

4.7. Category G
He cannot recall whether there were any differences.

5. **Finally: One possibility would be to create "analogy questions" which are independent of the language use. We've done some experiments in that regard, but we've come to the conclusion, that solving these questions depends more on correctly interpreting the prompt and on the students' understanding of logic, than effectively their grasp on programming concepts. Would you have any suggestions?**
The instructor mentioned the issue that certain analogies are dated (such as using a phone book in an analogy for a search algorithm).

4.3 I3

Date: 26 July 2017

1. **Introductory notes**
As the instructor was already familiar with the research this step was skipped.

2. ***May we record this interview?***
The instructor conceded to the recording.

3. ***Have you taught MC102 or other introductory-level CS courses in other programming languages besides Python? If so, which?***
The instructor taught MC102 in C in other terms, and in Python in the second term of 2015.

4. ***We've separated CS1 topics in 7 categories, having identified 33 misconceptions altogether. These categories are (A) function parameter use and scope (B) Variables,***

identifiers, and scope (C) Recursion (D) Iteration (E) Structures (F) Pointers (G) Boolean expressions.

4.1. **Regarding Category A (Function parameter use and scope), which misconceptions did you find your students had? Do you believe these are the same as we've found in C? To what point are we able to generalize (find common ground between C and Python)?**
The instructor does not recall doubts the students might have had in that respect. No nested functions were taught in the course, nor was the use of argument names.

4.2. **Question 4.1 repeated for category B**
The instructor could not point out significant differences between C and Python. He recognized Python's dynamic typing might create problems in theory, but was unable to recall whether those problems manifested themselves in his teaching experience.

4.3. **Category C**
He agreed it is a general problem, but could not recall specific issues.

4.4. **Category D**
Once again, he said the problems are generic and that students would have difficulties in C and in Python. Students did not learn how to use the `range()` constructor with more than two arguments.

4.5. **Category E**
The instructor said that objects were used as a replacement for teaching `structs` in C. He does not agree with this approach and feels it should be reconsidered in the event that Python is once again used in MC102.

4.6. **Category F**
Pointers were not taught.

4.7. **Category G**
He does not recall whether there were differences.

5. ***Finally: One possibility would be to create "analogy questions" which are independent of the language use. We've done some experiments in that regard, but***

we've come to the conclusion, that solving these questions depends more on correctly interpreting the prompt and on the students' understanding of logic, than effectively their grasp on programming concepts. Would you have any suggestions?
He agreed these questions should be implemented, but could not suggest examples.

5. Misconceptions

In Section 3 we expounded the process that resulted in the selection misconception hypotheses listed on the rightmost column of Table 11. We then proceeded to describe these potential in full, following the Anti-pattern [5] method. The rationale of this method is presenting a user with a fallacious solution that may be applicable, and then offer an alternative correct solution, so as to assist the user in avoiding the same incorrect solution in the future.

The template used to describe the hypothetical misconceptions in this Section comprises the following:

- **Code:** a unique key identifying the misconception within the scope of the Computer Science CI. All Python misconception codes are in the format *PX.n*, where *P* represents the Python language, *X*, the category of misconceptions, and *n = 1, 2, 3,* ... a sequential identifier.
- **Name:** a short descriptor for the misconception, aimed at summarizing the mistake in a few words, *e.g.* "Parameter value set by external source."
- **Description:** a more detailed explanation of the patterns described by the misconception.
- **Example:** an example of programming code in which the misconception happens. In other words, an excerpt of **incorrect** code reflecting the misconception.
- **Rationale:** the reason why we believe the misconception happens.
- **Consequences:** what may or will happen, should the misconception arise.
- **Detection:** where and how the misconception might appear.
- **Improvement:** how the misconception may be prevented.

5.1 Category A: Function Parameter Use and Scope

PA.1

Code:	PA.1
Name:	Parameter value set by an external source.
Description:	Parameter value is set by an external source outside the function scope.

Example:	`def func (n):`
	`    n = eval(input())`
	`    (...)`

Rationale:	Students fail to acknowledge that, when a function is defined for one or more parameters, then those parameters are set according to the arguments that are passed when that function is called. They find it therefore necessary to set one or more parameter values through an external source, such as a call to `input()`.
Consequences:	The correct parameter value will be lost.
Detection:	**Where:** • On any line of a function that receives a parameter.
	How: • The parameter value is wrongly modified.
Improvement:	Students should be oriented, as an exercise, to verify, using `print()`, for instance, the value of some parameter inside a function. If they are convinced the parameter's value has already been set, they are unlikely to reassign its value.

PA.3

Code:	PA.3
Name:	Attempt to access parameter from outside scope
Description:	Assumption that function parameters could be accessed anywhere in the program, regardless of scope.

Example:	`1. def func1 (n):`
	`2.     n = n + 5`
	`3.     return n`
	`4.`
	`5. def func2 (x):`
	`6.     return x + n`
	
	There is an error on line 6, which contains an attempt to access the parameter `n` from `func1` from within `func2`, which is outside the scope in question.

Rationale:	Students consider the scope of parameter variables to be global so that they could be accessed from anywhere in the code.
Consequences:	- The Python interpreter would raise a `NameError` exception, or; - If there is a variable (local or global), at any place in the code, which has the same name of some parameter, the student could consider the program will use that parameter value instead of the local value, leading to unexpected behavior.
Detection:	**Where:** • Inside a function, when a parameter variable outside the function scope is used.
	How: • Student wants to access a parameter variable not valid in the current function scope.
Improvement:	Students should be oriented, through examples and exercises, to understand the difference between local and global variable scope.

PA.5

Code:	PA.5
Name:	Function return value is not handled by the caller function
Description:	Assumption that return values are automatically handled by the caller function.

Example:	`1. def squareOf(n):`
	`2.     return n * n`
	`3.`
	`4. a = 5`
	`5. squareOf(a)`
	`6. print(a)`
	
	There is an error on line 5. The correct code would be:
	`5. a = squareOf(a)`

Rationale:	Students believe the program is intelligent, *i.e.*, it will understand the semantics related to a function call, automatically adjusting the logic and results.
Consequences:	The function's return value will not be received by the calling instance, rendering the function useless.
Detection:	**Where:** • When a function that returns a value is called.
	How: • The returned value is not correctly handled.
Improvement:	Students should be oriented to always check whether a non-void function's return value is being properly assigned.

5.2 Category B: Variables, Identifiers, and Scope

PB.1

Code:	PB.1
Name:	Attempt to access local variables from outside scope.
Description:	Out of scope assignment, considering or classifying local variables as if they were global variables.

Example:	`1. def findSmallest(A):`
	`2.     smallestIndex = 0`
	`3.     for i in range(1, len(A)):`
	`4.         if A[i] < A[smallestIndex]:`
	`5.             smallestIndex = i`
	`6.     return A[smallestIndex]`
	`7.`
	`8. # Some non-empty list A is instanced`
	`9. v = findSmallest(A)`
	`10. print("The smallest value is %d, at index %d." % (v, smallestIndex))`
	
	In this example, on line 10 the variable `smallestIndex` is referenced, even though it is local to the function `findSmallest`. A `NameError` exception would be raised.
	For this code to work, the `findSmallest` function should be modified to return the index instead of, or in addition to, the value, or the index could be retrieved with the line
	
	`smallestIndex = A.index(v)`
	
	inserted before line 10.

Rationale:	Students believe the scope of local variables is global, so they could be accessed at any point of the code.
Consequences:	- The program will raise an exception, or; - If there is a local variable in some function that has the same name of another local variable (in a different function), the student might assume these variables are the same, leading to unexpected behavior.
Detection:	**Where:** • In any function that has local variables.
	How: • The function tries to access a local variable declared in another function.

Improvement:	Students should be guided about the existing variable scopes in the Python language and how they work. Examples to show this might include comparisons between similar codes with different variable names, stressing that the chosen names make no difference regarding their scopes.

PB.3

Code:	PB.3
Name:	Parameter mistaken for same-name variable outside the function.
Description:	The wrong variable is accessed when a parameter has the same name of some local variable in the caller function.

Example:	`1. def add (a, b):`
	`2.     sum = a + b`
	`3.`
	`4. sum = 0`
	`5. a = 10`
	`6. b = 12`
	`7. add(a, b)`
	`8. print(sum)`
	
	It is assumed that the function `add` is altering the value of the variable `sum` that was instanced on line 4, but it is in fact creating a separate, local instance of sum that is lost when the function ends. Line 8 will print 0 and not 22. Before line 2, there should have been a declaration `global sum`, to inform the interpreter to use the global variable `sum`, rather than create a local one with the same name.

Rationale:	Students consider variables with the same name to refer to the same variable and memory values.
Consequences:	The program will not run as intended, leading to unexpected behavior.
Detection:	**Where:** • In any program that has at least 2 variables with the same name.
	How: • Student access or assign a value to some variable, but in fact he or she is accessing another variable in the memory.
Improvement:	Students should be oriented to understand that, as the Python interpreter does not require variables to be declared prior to assignment, it will create new variables as needed, prioritizing a local scope over a more global one. Thus they should take into consideration whether an external variable is visible from within a given context, and, if necessary, direct the interpreter to refer to that variable. Ideally, students should be taught to refrain from repeating variable names in different scopes, so as to avoid this type of issue altogether.

PB.5

Code:	PB.5
Name:	`for` statements considered as a function; iterating variable considered local
Description:	Variables created during `for` statement are considered to be part of their own lower-level scope and separate from the scope in which the loop is declared.

Example:	`1. i = 10`
	`2. a = 0`
	`3. for i in range(20):`
	`4.     a += 1`
	`5. print(a + i)`
	
	It is considered that the `for` statement creates a separate instance of `i` for the sole purpose of iterating over the specified loop, and that the global variable `i` remains unaffected. Thus, line 5 will print 40 and not 30, as was expected.
	
	The correct code, in this example, would assign different identifiers for these variables:
	
	`1. n = 10`
	`(...)`
	`5. print(a + n)`

Rationale:	Students consider `for` statements to be functions in their own right, with their own separate local scopes.
Consequences:	The program will not run as expected, leading to unexpected behavior.
Detection:	**Where:** • Whenever `for` statements are used.
	How: • The program logic is built upon the assumption that variables created in loop statements do not affect variables elsewhere in the scope.
Improvement:	Students should be oriented about the existing variable scopes in the Python language and how they work. Debugging and assertive programming should also be employed.

PB.6

Code:	PB.6
Name:	Function return value automatically updates when variable previously passed as arguments are changed.
Description:	A function is called and variables are passed as its arguments. The return value is assigned to a new variable. This variable is expected to automatically update when the variables passed as arguments have new values assigned to them.

Example:	`1. def area(base, height)`
	`2.     return base * height / 2`
	`3.`
	`4. base = 10`
	`5. height = 5`
	`6. a = area(base, height)`
	`7. base = 12`
	`8. print(area)`
	
	On line 8, the value of `area` is still 7.5, assigned on line 6, and does not change when `base` is given a new value in line 7.

Rationale:	Students believe that the Python interpreter will automatically update a variable when its initial assignment is the return value of a function called with previously created variables passed as arguments.
Consequences:	Students will fail to update the values of a given variable when the other variables change.
Detection:	**Where:** • Any program where variables receive return values of functions.
	How: • Student fails to update the value of a dependent variable
Improvement:	Students should be oriented to verify which variables depend on the values of other variables and check whether they are being updated accordingly.

5.3 Category C: Recursion

PC.1

Code:	PC.1
Name:	Wrong formula used to calculate the return value of a recursive function.
Description:	A recursive function returns a wrong value, leading to an improper sequence of recursive calls.

Example:	`1. # My recursive function`
	`2. def recursive (a):`
	`3.    if a == 0:`
	`4.       return 1`
	`5.`
	`6.    return recursive(a + 1)`
	`7.`
	`8. r = recursive(1)`
	
	In this example, the wrong return value in line 6 would lead to an infinite loop and consequently a `RecursionError` exception. The correct code to be replaced in line 6 is:
	
	`6.     return recursive (a - 1)`

Rationale:	This misconception relates to the lack of understanding about how recursion works in Python, specifically how recursion instances are stored in and retrieved from the stack.
Consequences:	The recursive function will present abnormal behavior, leading to an exception or wrong number of iterations. The program will not run as expected.
Detection:	**Where:** • Within the recursion function, specifically in the recursive call line.
	How: • The recursion never converges to its base case (infinite loop) or, if it does converge, the number of recursive calls is incorrect.

Improvement:	Students should understand that, in Python, recursion is handled by a stack. In this way, each recursion's instance is stored in the current topper position of the stack. To complete the recursion, a base case should be satisfied, *i.e.*, at some point the stack must cease to increase and each stored instance retrieved and resumed *(last in, first out scheme*). For example, this understanding could be reached through debugging techniques, such as the use of the `print` function to identify the position of each instance in the stack.

PC.2

Code:	PC.2
Name:	No recursive call
Description:	A presumably recursive function does not contain a call to itself anywhere in its code.
Example:	`// My recursive function` `def recursive (a):` `    if a == 0:` `        return 1` ` ` `r = recursive(10)`
Rationale:	Students have a weak grasp on the concept of recursion itself, which leads them to believe the function does not need to contain a recursive call.
Consequences:	As the presumably recursive function contains only the stopping condition, no recursive calls will be made, leading to incorrect logic in the program's execution.
Detection:	**Where:** • In any recursive function.
	How: • The recursive function lacks a recursive call.
Improvement:	Students should understand that, in Python, recursion is handled by a stack. In this way, each recursion's instance is stored in the current topper position of the stack. To complete the recursion, a base case should be satisfied, *i.e.*, at some point the stack must cease to increase and each stored instance retrieved and resumed *(last in, first out scheme*). For example, this understanding could be reached through debugging techniques, such as the use of the `print` function to identify the position of each instance in the stack.

PC.3

Code:	PC.3
Name:	Function does not terminate at the base case.
Description:	A recursive function lacks a base case (stop condition).

Example:	`// My recursive function`
	`def recursive (a):`
	`    return recursive(a - 1)`
	
	`r = recursive(10)`
	
	In this example, the misconception would lead to an infinite loop and consequently a `RecursionError` exception.

Rationale:	The recursive function is written upon the assumption that it will stop at a given time; the student lacks the required knowledge to define stop conditions within the recursive function code.
Consequences:	The program will not stop until it raises a `RecursionError` exception.
Detection:	**Where:** • Within recursive functions.
	How: • The recursive function does not have a stop condition.
Improvement:	Students can be oriented on how the recursive calls mechanism works (aspects such as the memory stack and how the interpreter handles recursive calls), so that they understand the cruciality of stop conditions.

5.4 Category D: Iteration

PD.1 -

Code:	PD.1
Name:	Improper update of a loop counter
Description:	Loop counter is not properly updated, leading to an improper number of executions.

Example:	`1. i = 0`
	`2. sum = 0`
	`3. while i < 10:`
	`4.     sum = sum + i`
	`5.     i = 1`
	
	The issue is shown on line 5 (the correct counter update would be `i += 1`). This program, run as is, will enter an infinite loop.

Rationale:	Students fail to understand how the loop counter should be changed (increased or decreased) to reach the number of iterations desired.
Consequences:	Infinite loop or incorrect results, if the output uses values calculated inside the loop.
Detection:	**Where:** • In any loop structure (`for` or `while`)
	How: • The loop counter is not updated or is incorrectly updated.
Improvement:	Students should be able to check (*e.g.* through a debugging tool or the use of `print` statements) the counter value at each iteration. In this way, they would be able to ascertain whether the counter is being updated as expected.

PD.2 -

Code:	PD.2
Name:	Use of loop result before loop completes.
Description:	Partial results found during loop iterations are used or considered as if they were final.

Example:	`1. sum = 0`
	`2. for i in range(10):`
	`3.     sum = sum + i`
	`4.     print("The sum is %d" % sum)`
	`5.`
	`6. print("The sum is %d" % sum)`
	
	For each iteration the partial result is needlessly printed (line 4).

Rationale:	Students feel inclined to group all code related to a given loop inside the body of that loop.
Consequences:	The loop will not produce the expected result.
Detection:	**Where:** • Wherever a loop is used and there are operations related to its final result.
	How: • Students include in the body of the loop pieces of code that are meant to use its final result.
Improvement:	Students should be oriented to check each instruction within a loop and determine whether it depends on a partial result or a final result, moving the latter outside of the body.

PD.3 -

Code:	PD.3
Name:	Loop iterates the correct number of times, but over the incorrect range.
Description:	Improper initialization of the loop counter or improper construction of its stop condition, leading to an unexpected number of loop iterations.

Example:	Example:
	Write a program that prints the sum of all numbers from 1 to 9:
	
	`1. sum = 0`
	`2. for i in range(9):`
	`3.     sum = sum + i;`
	`4.`
	`5. print("The sum is %d " % sum)`
	
	In this case, the `range` object instanced on line 2 returns a sequence of numbers from 0 to 8, rather than from 1 to 9. It should have been initialized as `range(1,10)`.

Rationale:	Students does not understand how to construct the loop structure (counter initialization or stop condition) to support a specific number of iterations.
Consequences:	The loop will iterate over the wrong interval, leading to unexpected program behavior.
Detection:	**Where:** • When a loop is defined.
	How: • A `range` object is improperly instanced, through lack of specification of a starting index, and / or through misunderstanding of the behavior of the stop index, or • A while-loop is not properly initialized.
Improvement:	Students should be oriented to check the range over which the loop will iterate and initialize the `range` object or while-loop accordingly—they might wish to print out the indexes once to see if the loop has been initialized appropriately.

PD.4

Code:	PD.4
Name:	Wrong flow control in a loop
Description:	The number of times a loop is executed is wrong.

Example:	Write a program that prints all integers, from 0 to 9:
	`sum = 0`
	`for i in range(9):`
	`    print(i)`
	
	`print ("The sum is %d" % sum)`

Rationale:	Students does not understand how to construct the loop structure (counter initialization or stop condition) to support a specific number of iterations.
Consequences:	The loop behavior will not be as expected. The number of iterations will be greater or lower than it should be.
Detection:	**Where:** • *For* iterations: The issue can be found in how the range object is instanced, specifically the stop condition, and the starting index and step attribute if either is declared. • *While* iterations: The issue can be found before the *while* structure declaration, when the loop control variable is initialized; it also can be found in the *while* condition, that determines the condition to the loop be executed and; it can be found in the *while* conditional code, that can have a wrong control variable increment – or even is absence.
	How: The detection of this misconception depends upon the analysis of the intended use of the loop, *i.e.*, determining what the loop is supposed to do (or calculate) and how many times it is expected to iterate.
Improvement:	Students should be oriented to check if the loop control variable has been correctly initialized and incremented. Also, they should check whether the loop stop condition was correctly defined. In order to ascertain how many times the loop is executed, students could insert a *print* statement inside the loop block, printing out the value of the loop control variables.

PD.5 -

Code:	PD.5
Name:	No loop, only one simple iteration
Description:	No loop structure is declared, where one would be necessary for the program to function as expected.

Example:

```
1.  def isDivisibleBy (n, x):
2.     if n % x == 0:
3.        return True
4.     else:
5.        return False
6.
7.  x = eval(input())
8.  isPrime = isDivisibleBy(x, 1)
9.
10. if isPrime:
11.     print(" Number %d is prime" % x)
12. else:
13.     print(" Number %d is not prime" % x)
```

In this example, the call to `isDivisibleBy` is made only once (line 8), without any iteration.

Rationale:	The call to `isDivisibleBy` is written upon the assumption that the loop statement and its definitions will be automatically imposed by the Python interpreter.
Consequences:	In most of the cases the loop absence will cause the program to fail to work as expected. In specific situations (when the loop would iterate only once) the program will work successfully.
Detection:	**Where:** • Anywhere throughout the code where the intention of a repetition structure can be inferred.
	How: • The code logic requires some block to be repeated several times. However, the program does not have any repetition loop statement.
Improvement:	Students should be oriented on how loop statements are handled by the Python interpreter and that the use of a loop command (be it `for` or `while`) is required to that end.

PD.6

Code:	PD.6
Name:	Loop construction does not consider the program logic and its connection with other parts of the code.
Description:	Although the loop is internally well constructed, there is a logic error when the big picture is analyzed, *i.e.*, how the loop interacts with the code before and after it.

Example:

```
1. def isDivisibleBy (a, b):
2.    if a % b == 0
3.       return True
4.    else:
5.       return False
6.
7. x = eval(input())
8. foundDivisible = True
9. c = 2
10. while not foundDivisible and c < x:
11.     foundDivisible = isDivisibleBy(x, c)
12. c += 1
13.
14. if not foundDivisible:
15.     print("Number %d is prime" % x)
16. else:
17.     print("Number %d is not prime" % x)
```

The `while` conditional expression on line 12 is evaluated as false on the first attempt, leading to a program that classifies all numbers as prime. The correct code is:

```
8.  foundDivisible = False
```

Another variation for this example would be if the line 14 contained the following code:

```
14.  if foundDivisible
```

In this situation, even if the loop correctly identifies whether a number is prime, the loop output (`foundDivisible` variable) is incorrectly used by other program parts.

Rationale:	Students build the loop considering it is an independent part of the code, unrelated to any preceding or succeeding code.
Consequences:	The program will not behave as expected, leading to logic errors.
Detection:	**Where:** • Whenever a loop is used within the code

	How: • If considered independently, the loop is well constructed. However, when considering the code surrounding loop, there is a logic error in the data that is used by the loop (input) or generated by the loop (output) and used in other parts of the code.
Improvement:	Students should be oriented to understand the loop is not an independent entity, thus being influenced by previous code and also influencing the code that is after it.

PD.7

Code:	PD.7
Name:	A `for` loop that iterates over elements is treated as if iterating over indices.
Description:	The for loop is declared without a range instance (e.g. iterating over elements of a list) but contains code that uses the iterator as a subscript in bracket notation.

Example:	`1. # A is a non-empty list object.`
	`2. for i in A:`
	`3.     print(A[i])`
	
	In this example, line 3 will request the element at index `i` of `A`. If `A` contains integers, then the values stored in `A` will be used as indices, leading to unpredictable behavior and possible an `IndexError` exception. If `A` contains values of a non-integer type, a `TypeError` exception will be raised.

Rationale:	Students assume the Python interpreter will immediately infer whether they need the indices of a range or the values of a list.
Consequences:	The program will have unexpected behavior or be interrupted by an uncaught exception.
Detection:	**Where:** • Whenever a `for` loop is used within the code
	How: • The `for` loop is declared without instancing or using a range object, but the iterator variable is used as an index. Note that there are specific cases where this may be appropriate (say, in a counting sort).
Improvement:	Students should be oriented to understand that Python `for` loops always iterate over the elements of whatever is specified after `in`, and if they require a sequence of numbers, they must use an instance of `range` (or other object as appropriate).

PD.8

Code:	PD.8
Name:	A for loop that iterates over a range is treated as if iterating over elements.
Description:	The `for` loop is declared with a range instance (e.g. iterating over elements of a list) but contains code that uses the iterator as direct reference to an element.

Example:	`1. # A is a non-empty list object.`
	`2. for i in range(len(A)):`
	`3.     print(i)`
	
	In this example, line 3 will simply print out the index `i`, which will contain a value between 0 and `len(A) - 1`. The actual values stored in A will not be accessed.

Rationale:	Students assume the Python interpreter will immediately infer whether they need the indices of a range or the values of a list.
Consequences:	The program will not behave as expected.
Detection:	**Where:** • Whenever a `for` loop is used within the code
	How: • The for loop is declared instancing a range object, but the index itself is not passed to the container object (e.g. a list) to retrieve the desired value where it would be appropriate to do so.
Improvement:	Students should be oriented to understand that the `range` class returns a simple sequence of integers, and that these integers must then be passed as subscripts or other means to retrieve the desired values.

5.5 Category G: Boolean Expressions

PG.1

Code:	PG.1
Name:	Incorrect precedence for Boolean operators.
Description:	Order of precedence of Boolean expression is different than the one was intended.

Example:	Consider the following definition: *A leap year is every year that is divisible by 4, with the exception of century years (i.e. years ending in 00), which will only be leap years if they are also divisible by 400). Write a function that takes an argument* y *containing an integer representing a year, and returns* True *or* False *specifying whether that year is a leap year.*
	
	`1. def isLeapYear(y):`
	`2.     if (y % 400 == 0 or y % 4 == 0) and y % 100 != 0:`
	`3.         return True`
	`4.     else:`
	`5.         return False`
	
	Line 2 contains an error. The correct expression would be
	
	`2.     if y % 400 == 0 or y % 4 == 0 and y % 100 != 0:`

Rationale:	Students either consider that Boolean expressions can be directly transcribed from English, or fail to examine the correct order of precedence followed by the Python interpreter.
Consequences:	The expression will not be evaluated correctly and the program will behave in an unexpected manner.
Detection:	**Where:** • Whenever Boolean expressions are used, in particular those involving more than one Boolean operator.
	How: • Students either fail to parenthesize the expression, or do so inappropriately.
Improvement:	Students should be oriented to review the order of precedence of Python operators and, when writing Boolean expressions, to ascertain that they will be evaluated by the program in the desired order.
Feedback:	Remember that, in Python, there is an order in which Boolean expressions are evaluated. Parenthesized expressions are evaluated first; then, in this order, the `not`, `and`, and `or` operators.

PG.2

Code:	PG.2
Name:	Nested if-statements where a single Boolean expression could have been used.
Description:	A Boolean expression is written as an `if-else` nested sequence.

Example:	`result = 0`
	`if A:`
	`    if B:`
	`        result = 1`
	`    elif C:`
	`        result = 1`
	`if result:`
	`    # Do something`
	
	The same code above should be written as:
	
	`if A and (B or C):`
	`    # Do something`

Rationale:	Students do not know how to evaluate multiple-variable Boolean expressions, and so they build a sequence of `if` and `else` statements, each containing a single variable to be evaluated.
Consequences:	Although the program will still work correctly with a sequence of `if` and `else` statements (*i.e.* this is not an error), longer Boolean expressions will lead to needlessly complex code, which might prove difficult to read, understand, and debug.
Detection:	**Where:** • Whenever a Boolean expression must be evaluated within the code.
	How: • The Boolean expression is evaluated through a sequence of `if` and `else` statements.
Improvement:	Students should be oriented to understand how Boolean expressions are evaluated in Python language, and also how to use the proper operators (`or`, `and`, `not`, and parentheses), to build these expressions. In addition, the equivalence to nested if-statements could be explored.
Feedback:	Writing a tree of if-else statements where a single Boolean expression could have been used can cause your code to be difficult to read and excessively long. While this code will work correctly, it is considered good programming practice to learn to build and use complex Boolean expressions when possible.

PG.4

Code:	PG.4
Name:	Attempt to evaluate a Boolean expression through loop iterations.
Description:	A loop statement is used where a Boolean expression would need to be evaluated only once.

Example:	Consider the variables a, b, and c, of type bool, which represent statements that can be true or false. Write a Boolean expression that evaluates whether at least two statements are true:
	
	`(...)`
	`while a and b or c:`
	`    # Do something`
	`(...)`
	
	The correct statement is
	
	`(...)`
	`if a and b or a and c or b and c:`
	`    # Do something`
	`(...)`

Rationale:	Students do not know how to construct a Boolean expression composed of a sequence of multiple statements. They do, however, have the notion that the desired Boolean expression would comprise a sequence of statements that have some correlation among them. Therefore, students conclude that a loop iteration will somehow support this approach.
Consequences:	The Boolean expression will not be correctly evaluated and the program will not work properly. Also, as the loop contains logical errors, there is a chance that the loop condition will always evaluate as true, leading to an infinite loop error.
Detection:	**Where:** • Whenever a Boolean expression with multiple statements must be evaluated within the code.
	How: • A logic problem is solved through a sequence of loop iterations, rather than a Boolean expression.
Improvement:	Students should be oriented to understand how Boolean expressions are evaluated in Python language, and also how to use the proper operators (`or`, `and`, `not`, and parentheses), to build these expressions.
Feedback:	You've attempted to use a loop to construct and evaluate a Boolean expression comprising multiple variables. A loop is not necessary here--the Boolean expression only needs to be evaluated once. If the loop is

	used and the expression is evaluated as true, the program will enter an infinite loop.

5.6 Category H: Use and Implementation of Classes and Objects

PH.1 -

Code:	PH.1
Name:	Attempt to invoke method outside its class.
Description:	Attempt to invoke an existing method from a different class from the one where it is implemented.

Example:	`1. def addZCoordinate(xyCoords: tuple, zCoord: int):`
	`2.     xyCoords.append(zCoord)`
	`3.     return xyCoords`
	
	In this example, line 2 is written on the assumption that a value can be added to a tuple through the `append` method, which is declared in the `list` class. Line 2 will raise an `AttributeError` and the program will not execute.
	
	The correct code for line 2 would be
	
	`2.     xyCoords = xyCoords + (zCoord,)`
	
	in which a new three-element tuple is instanced and can then be used as a return value.

Rationale:	Students consider that methods internal to a class can be universally invoked from other types of similar character.
Consequences:	An `AttributeError` exception will be raised and the program will halt if it is not caught.
Detection:	**Where:** • Whenever methods are invoked, especially those of built-in Python types.
	How: • A method that is not declared in a class is invoked from it, usually through the dot operator.
Improvement:	Students should be oriented to understand the implementation of the specific classes they want to use.

PH.2 -

Code:	PH.2
Name:	Treating method that returns a new instance of the same object as one that changes the instance itself.
Description:	A method that operates somehow on the caller instance and then returns a new instance with the result of the operation is not attributed to a variable.

Example:	`1. nameStatement = "Hello my name is Dax."`
	`2. nameStatement.replace("Dax","Zax")`
	
	In line 2, a new string containing "`Hello my name is Zax.`" is returned, but it is not assigned to a variable and the result is lost. `nameStatement` will retain the initial string that was assigned.

Rationale:	Students consider that methods that operate on the contents of an instance will automatically change that instance.
Consequences:	The desired result is lost.
Detection:	**Where:** • Whenever a method is called upon to perform some operation on the contents of an instanced object. This is especially the case when `str` objects are in consideration, due to the immutability of strings in Python.
	How: • A method is called to generate a result based on the current value, but the result is not assigned to a variable.
Improvement:	Students should be oriented to understand the implementation of the specific classes they want to use. They should learn to check the documentation to understand the return values of methods.

PH.3 -

Code:	PH.3
Name:	Function call preceded by `def` keyword
Description:	A function is called following a `def` keyword, as if it were a function declaration.

Example:	`1. def addTwo(A):`
	`2.     for i in range(len(A)):`
	`3.         A[i] += 2`
	`4.`
	`5. A = [1,2,3,4,5]`
	`6. def addTwo(A)`
	`7. print(A)`
	
	Line 6 would generate a `SyntaxError`, as the Python interpreter expects a colon (:) at the end of a function definition. The correct code would be
	
	`6. addTwo(A)`

Rationale:	Students assume the `def` keyword to be necessary whenever a reference to a function is made, rather than only when the function is defined.
Consequences:	A `SyntaxError` exception will be raised.
Detection:	**Where:** • Where recently-defined functions are used.
	How: • Functions are called preceded by the `def` keyword.
Improvement:	Students should learn to differentiate the definition of a function from its later use.

PH.4

Code:	PH.4
Name:	Attribute invoked without being imported and with no class specified.
Description:	An attribute of a given class (and therefore not built-in) is invoked, without having previously been imported or without being called from a class or instance thereof.

Example:	`1. for i in range(10):`
	`2.     print("%.2f" % sqrt(i))`
	
	In this example, the `sqrt` method (presumably from the `math` module) is called without having previously been imported, leading to a `NameError` exception.
	
	This code will only function if, before line 1, the method is imported in a statement such as
	
	`from math import sqrt`

Rationale:	The Python interpreter is capable of calling an attribute simply by its name, without being told where to look for it.
Consequences:	A `NameError` exception.
Detection:	**Where:** • Whenever attributes (particularly non-built-in functions) are used
	How: • The attribute is called by its identifier, without being imported itself or called from an imported class.
Improvement:	Students should be oriented to study the built-in functions in Python and understand that other functions or attributes need to be imported.

PH.5 -

Code:	PH.5
Name:	Assigning value to method
Description:	An attempt is made to assign a value to the identifier of a method.

Example:	`1. A = [1,2,3,4,5]`
	`2. A.append = 6`
	
	In this example, line 2 will cause an `AttributeError` exception to be raised, as the `append` method is a read-only attribute of the `list` class.
	
	The correct code would be
	
	`2. A.append(6)`

Rationale:	Students mistake the use of methods for the use of variables and might assume the Python interpreter can easily switch between the relevant syntaxes.
Consequences:	An `AttributeError` exception is raised.
Detection:	**Where:** • Whenever methods are called from their classes.
	How: • Student assigns a value to the method identifier, rather than use the parentheses syntax to specify parameters.
Improvement:	Students should be oriented on the correct syntax for calling methods.

PH.6 -

Code:	PH.6
Name:	No `self` keyword to reference instance attributes.
Description:	Within the definition of a class, the attributes and methods of that class are called without the keyword `self` and the dot operator.

Example:	`1. class Dog:`
	`2.     sound = "Woof!"`
	`3.     def makeSound(self):`
	`4.         print(sound)`
	
	When line 4 is executed, the Python interpreter will raise a `NameError` exception, as it does not know what the sound identifier refers to. The correct code would be
	
	`4.         print(self.sound)`

Rationale:	Students consider that attributes of a class can be directly referenced from within the class definition, without use of the `self` keyword.
Consequences:	A `NameError` exception is raised.
Detection:	**Where:** • Whenever new classes are defined by students.
	How: • Student attempts to call a class attribute without preceding it with `self.`
Improvement:	Students should be oriented to understand the proper syntax for class definition and the situations when the use of `self` is applicable.

PH.7 -

Code:	PH.7
Name:	Attempt to call class attributes through subscripts.
Description:	An object of a class with no `__getitem__()` method is instanced, and then it is attempted to change the values of its attributes with bracket operators and indices, as if the object were a list or similar structure.

Example:

```
1. class Dog:
2.     def __init__(self, name, age, favoriteToy):
3.         self.name = name
4.         self.age = age
5.         self.favoriteToy = favoriteToy
6.
7. newDog = Dog("Rex", 9, "rubber ducky")
8. newDog[1] = 10
```

Line 8 will raise a `TypeError` exception, as `newDog` is an instance of the `Dog` class, which does not support indexing. The correct code would be

```
8. newDog.age = 10
```

Rationale:	Students consider that class attributes may be referenced and assigned through subscript indices (when no support for indexing has been implemented), particularly when said attributes are consistently defined in a specific order (*i.e.* in the constructor method).
Consequences:	A `TypeError` exception will be raised.
Detection:	**Where:** • Whenever new classes are designed and used.
	How: • An instance of that class is called using the bracket operator and an index, in an attempt to access an attribute.
Improvement:	Students should be oriented to understand how class attributes work and how they are different from list indices.

6. Conclusions and Future Work

From the beginning of the research steps covered in this report, our intent has been to delve into the process of learning introductory Computer Science in the Python language, in an effort to acquire useful data to inform the further development of a CS Concept Inventory in Python.

The hypothetical misconceptions described in Section 5 now require statistical validation with CS1 students who have taken their introductory courses in Python. This will be addressed in future works.

7. Acknowledgements

This research is supported by grant #2014/07502-4, São Paulo Research Foundation (FAPESP). Additional support was provided by the Brazilian Federal Agency for Support and Evaluation of Graduate Education (CAPES), the National Counsel of Technological and Scientific Development (CNPq) and the University of Campinas (Unicamp).

We would like to thank Dr. Breno de França, Dr. Marco Aurélio Gerosa, Dr. Eduardo Aranha, Dr. Heiko Hornung, Dr. Jacques Wainer, Dr. Tânia Caldas and Dr. Eduardo Xavier for the support and orientation on this research.

8. References

[1] CACEFFO, R.; WOLFMAN, S.; BOOTH, K. 2016. Developing a Computer Science Concept Inventory for Introductory Programming. *In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA, 364-369. DOI=http://dx.doi.org/10.1145/2839509.2844559

[2] CACEFFO, R. FRANÇA, B.; GAMA, G.; BENATTI, R.; APARECIDA, T.; CALDAS, T.; AZEVEDO, R. (2017) An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in C. In Technical Report 17-15, Institute of Computing, University of Campinas, SP, Brasil. 42 pages. October, 2017.

[3] CACEFFO, R.; GAMA, G.; AZEVEDO, R. (2018). Exploring Active Learning Approaches to Computer Science Classes. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18). ACM, New York, NY, USA, 922-927. DOI: https://doi.org/10.1145/3159450.3159585

[4] CACEFFO, R..; GAMA, G.; BENATTI, R.; APARECIDA, T.; CALDAS, T.; AZEVEDO, R. (2018) A Concept Inventory for CS1 Introductory Programming Courses in C. In Technical Report 18-06, Institute of Computing, University of Campinas, SP, Brasil. 107 pages. March, 2018

[5] Mohamed El-Attar and James Miller. 2006. Matching Antipatterns to Improve the Quality of Use Case Models. In *Proceedings of the 14th IEEE International Requirements Engineering Conference* (RE '06). IEEE Computer Society, Washington, DC, USA, 96-105. DOI=http://dx.doi.org/10.1109/RE.2006.42

[6] G. L. Herman, M. C. Loui, and C. Zilles. Creating the digital logic concept inventory. In Proceedings of the 41st ACM Technical Symposium on Computer Science Education, SIGCSE ’10, pages 102–106, New York, NY, USA, 2010. ACM

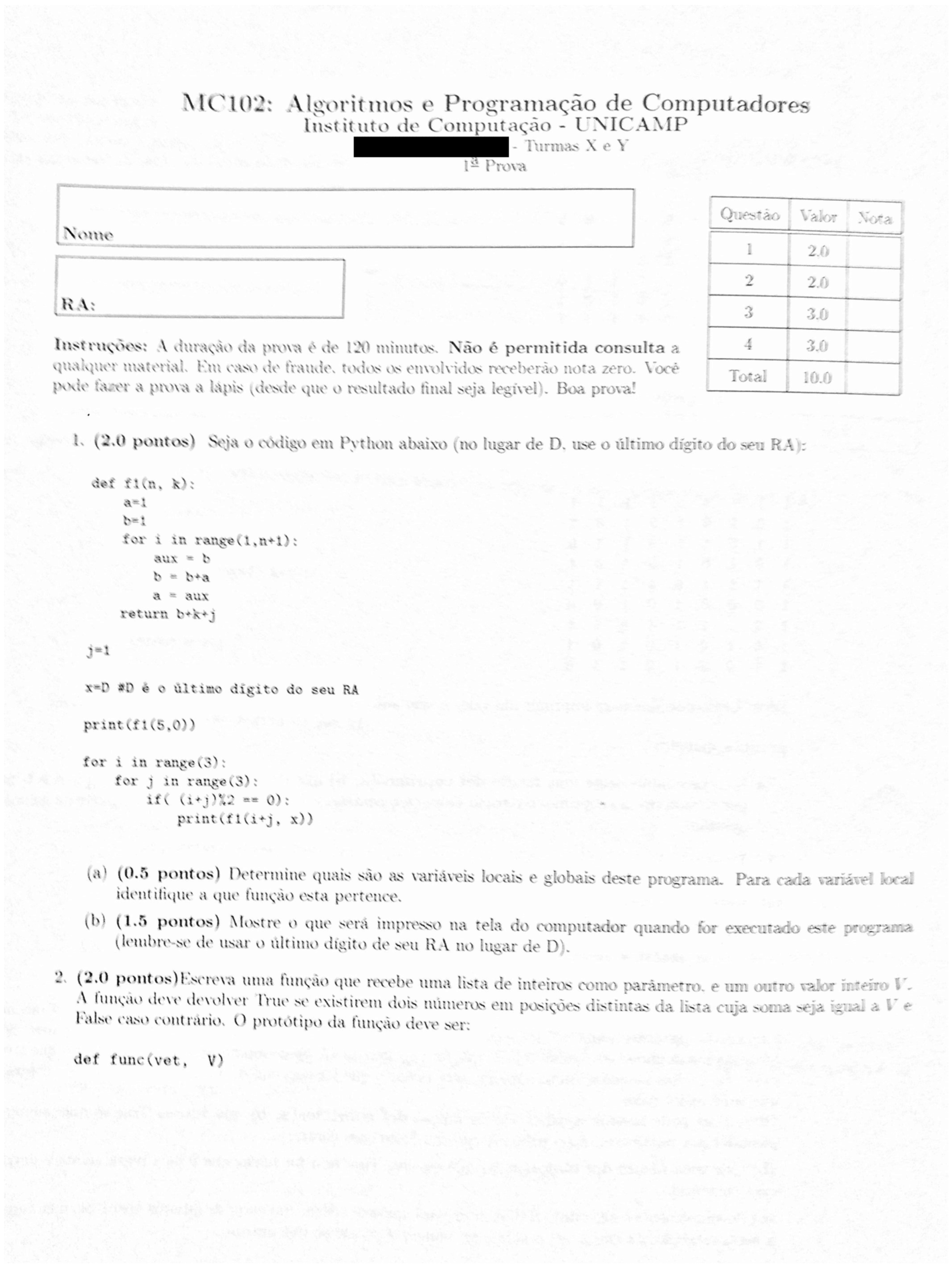
[7] ALMSTRUM, V.; HENDERSON, P.; HARVEY, V.; HEEREN, C.; MARION, W.; RIEDESEL, C.; SOH, L.; TEW, A. (2006) Concept Inventories in computer science for the topic discrete mathematics. In Proceedings of the Working group reports on ITiCSE on Innovation and technology in computer science education, ITiCSE-WGR ’06. ACM, New York, NY, USA, 132-145. DOI: https://doi.org/10.1145/1189215.1189182

Appendix A Exams in Portuguese

This appendix contains scans of the exams that were analyzed as described in Section 3. These are the original exams in Portuguese, as given to the students of each section. Identifying information of the instructors and / or students has been purposefully omitted. Subsections 8.A.1A.1, A.2, and A.3 contain the Midterm, Final, and Retake exams, respectively, of Instructor I1 for the 2nd Term 2015 section of MC102. Subsections A.4 and A.5 contain the Midterm and Final exams, respectively, of Instructor I2 for the 2nd Term 2016 section of MC102.

Full English translations of these exams are available in Appendix B.

A.1 MC102, 2nd Term 2015, Instructor I1, Midterm Exam



MC102: Algoritmos e Programação de Computadores

Instituto de Computação - UNICAMP

[redacted] - Turmas X e Y

1ª Prova

Nome

RA:

Questão	Valor	Nota
1	2.0	
2	2.0	
3	3.0	
4	3.0	
Total	10.0	

Instruções: A duração da prova é de 120 minutos. **Não é permitida consulta** a qualquer material. Em caso de fraude, todos os envolvidos receberão nota zero. Você pode fazer a prova a lápis (desde que o resultado final seja legível). Boa prova!

1. **(2.0 pontos)** Seja o código em Python abaixo (no lugar de D, use o último dígito do seu RA):

```
def f1(n, k):
    a=1
    b=1
    for i in range(1,n+1):
        aux = b
        b = b+a
        a = aux
    return b+k+j

j=1

x=D #D é o último dígito do seu RA

print(f1(5,0))

for i in range(3):
    for j in range(3):
        if( (i+j)%2 == 0):
            print(f1(i+j, x))
```

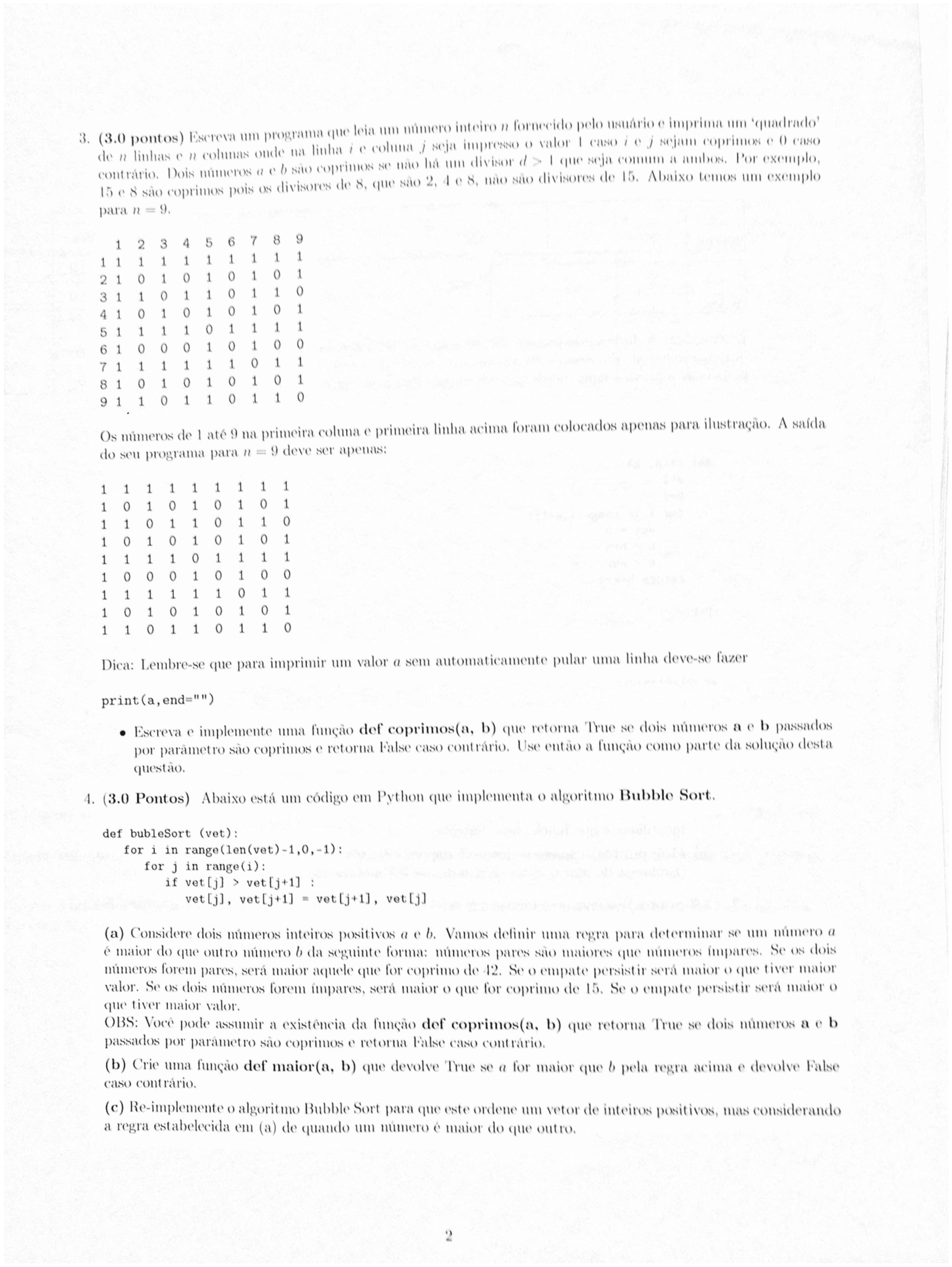
   (a) **(0.5 pontos)** Determine quais são as variáveis locais e globais deste programa. Para cada variável local identifique a que função esta pertence.

   (b) **(1.5 pontos)** Mostre o que será impresso na tela do computador quando for executado este programa (lembre-se de usar o último dígito de seu RA no lugar de D).

2. **(2.0 pontos)**Escreva uma função que recebe uma lista de inteiros como parâmetro, e um outro valor inteiro V. A função deve devolver True se existirem dois números em posições distintas da lista cuja soma seja igual a V e False caso contrário. O protótipo da função deve ser:

```
def func(vet,  V)
```

Figure 2: Midterm Exam, MC102 Section 2/2015, I1



3. **(3.0 pontos)** Escreva um programa que leia um número inteiro n fornecido pelo usuário e imprima um 'quadrado' de n linhas e n colunas onde na linha i e coluna j seja impresso o valor 1 caso i e j sejam coprimos e 0 caso contrário. Dois números a e b são coprimos se não há um divisor $d > 1$ que seja comum a ambos. Por exemplo, 15 e 8 são coprimos pois os divisores de 8, que são 2, 4 e 8, não são divisores de 15. Abaixo temos um exemplo para $n = 9$.

```
  1 2 3 4 5 6 7 8 9
1 1 1 1 1 1 1 1 1 1
2 1 0 1 0 1 0 1 0 1
3 1 1 0 1 1 0 1 1 0
4 1 0 1 0 1 0 1 0 1
5 1 1 1 1 0 1 1 1 1
6 1 0 0 0 1 0 1 0 0
7 1 1 1 1 1 1 0 1 1
8 1 0 1 0 1 0 1 0 1
9 1 1 0 1 1 0 1 1 0
```

Os números de 1 até 9 na primeira coluna e primeira linha acima foram colocados apenas para ilustração. A saída do seu programa para $n = 9$ deve ser apenas:

```
1 1 1 1 1 1 1 1 1
1 0 1 0 1 0 1 0 1
1 1 0 1 1 0 1 1 0
1 0 1 0 1 0 1 0 1
1 1 1 1 0 1 1 1 1
1 0 0 0 1 0 1 0 0
1 1 1 1 1 1 0 1 1
1 0 1 0 1 0 1 0 1
1 1 0 1 1 0 1 1 0
```

Dica: Lembre-se que para imprimir um valor a sem automaticamente pular uma linha deve-se fazer

```
print(a,end="")
```

- Escreva e implemente uma função **def coprimos(a, b)** que retorna True se dois números **a** e **b** passados por parâmetro são coprimos e retorna False caso contrário. Use então a função como parte da solução desta questão.

4. **(3.0 Pontos)** Abaixo está um código em Python que implementa o algoritmo **Bubble Sort**.

```
def bubleSort (vet):
   for i in range(len(vet)-1,0,-1):
      for j in range(i):
         if vet[j] > vet[j+1] :
            vet[j], vet[j+1] = vet[j+1], vet[j]
```

(a) Considere dois números inteiros positivos a e b. Vamos definir uma regra para determinar se um número a é maior do que outro número b da seguinte forma: números pares são maiores que números ímpares. Se os dois números forem pares, será maior aquele que for coprimo de 42. Se o empate persistir será maior o que tiver maior valor. Se os dois números forem ímpares, será maior o que for coprimo de 15. Se o empate persistir será maior o que tiver maior valor.
OBS: Você pode assumir a existência da função **def coprimos(a, b)** que retorna True se dois números **a** e **b** passados por parâmetro são coprimos e retorna False caso contrário.

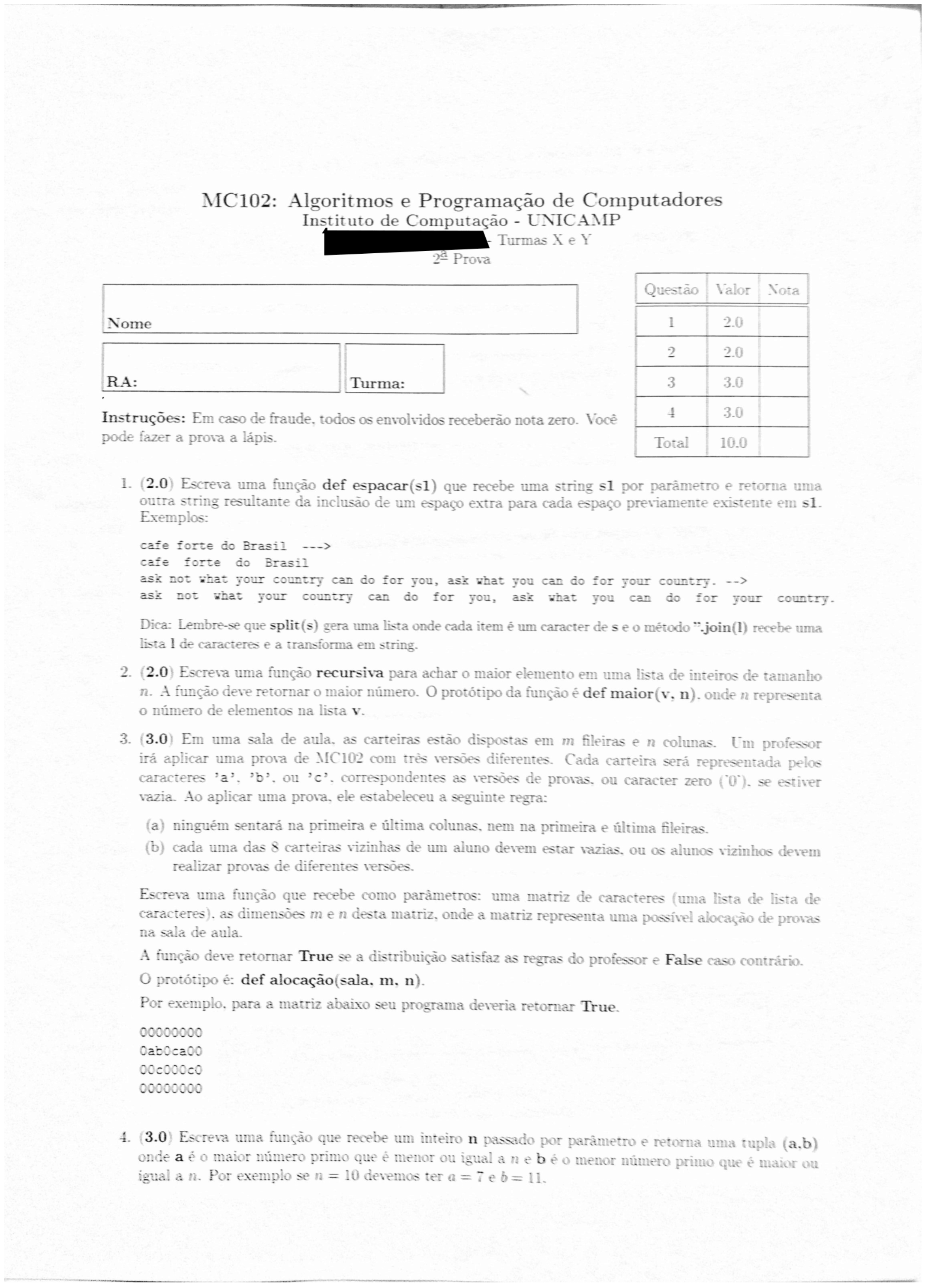
(b) Crie uma função **def maior(a, b)** que devolve True se a for maior que b pela regra acima e devolve False caso contrário.

(c) Re-implemente o algoritmo Bubble Sort para que este ordene um vetor de inteiros positivos, mas considerando a regra estabelecida em (a) de quando um número é maior do que outro.

2

Figure 3: Midterm Exam, MC102 Section 2/2015, I1, page 2

A.2 MC102, 2nd Term 2015, Instructor I1, Final Exam



MC102: Algoritmos e Programação de Computadores

Instituto de Computação - UNICAMP

[redacted] - Turmas X e Y

2ª Prova

Nome

RA: Turma:

Questão	Valor	Nota
1	2.0	
2	2.0	
3	3.0	
4	3.0	
Total	10.0	

Instruções: Em caso de fraude, todos os envolvidos receberão nota zero. Você pode fazer a prova a lápis.

1. (**2.0**) Escreva uma função **def espacar(s1)** que recebe uma string **s1** por parâmetro e retorna uma outra string resultante da inclusão de um espaço extra para cada espaço previamente existente em **s1**. Exemplos:

```
cafe forte do Brasil  --->
cafe  forte  do  Brasil
ask not what your country can do for you, ask what you can do for your country. -->
ask  not  what  your  country  can  do  for  you,  ask  what  you  can  do  for  your  country.
```

   Dica: Lembre-se que **split(s)** gera uma lista onde cada item é um caracter de **s** e o método **''.join(l)** recebe uma lista **l** de caracteres e a transforma em string.

2. (**2.0**) Escreva uma função **recursiva** para achar o maior elemento em uma lista de inteiros de tamanho n. A função deve retornar o maior número. O protótipo da função é **def maior(v, n)**, onde n representa o número de elementos na lista **v**.

3. (**3.0**) Em uma sala de aula, as carteiras estão dispostas em m fileiras e n colunas. Um professor irá aplicar uma prova de MC102 com três versões diferentes. Cada carteira será representada pelos caracteres `'a'`, `'b'`, ou `'c'`, correspondentes as versões de provas, ou caracter zero ('0'), se estiver vazia. Ao aplicar uma prova, ele estabeleceu a seguinte regra:

   (a) ninguém sentará na primeira e última colunas, nem na primeira e última fileiras.

   (b) cada uma das 8 carteiras vizinhas de um aluno devem estar vazias, ou os alunos vizinhos devem realizar provas de diferentes versões.

   Escreva uma função que recebe como parâmetros: uma matriz de caracteres (uma lista de lista de caracteres), as dimensões m e n desta matriz, onde a matriz representa uma possível alocação de provas na sala de aula.

   A função deve retornar **True** se a distribuição satisfaz as regras do professor e **False** caso contrário.

   O protótipo é: **def alocação(sala, m, n)**.

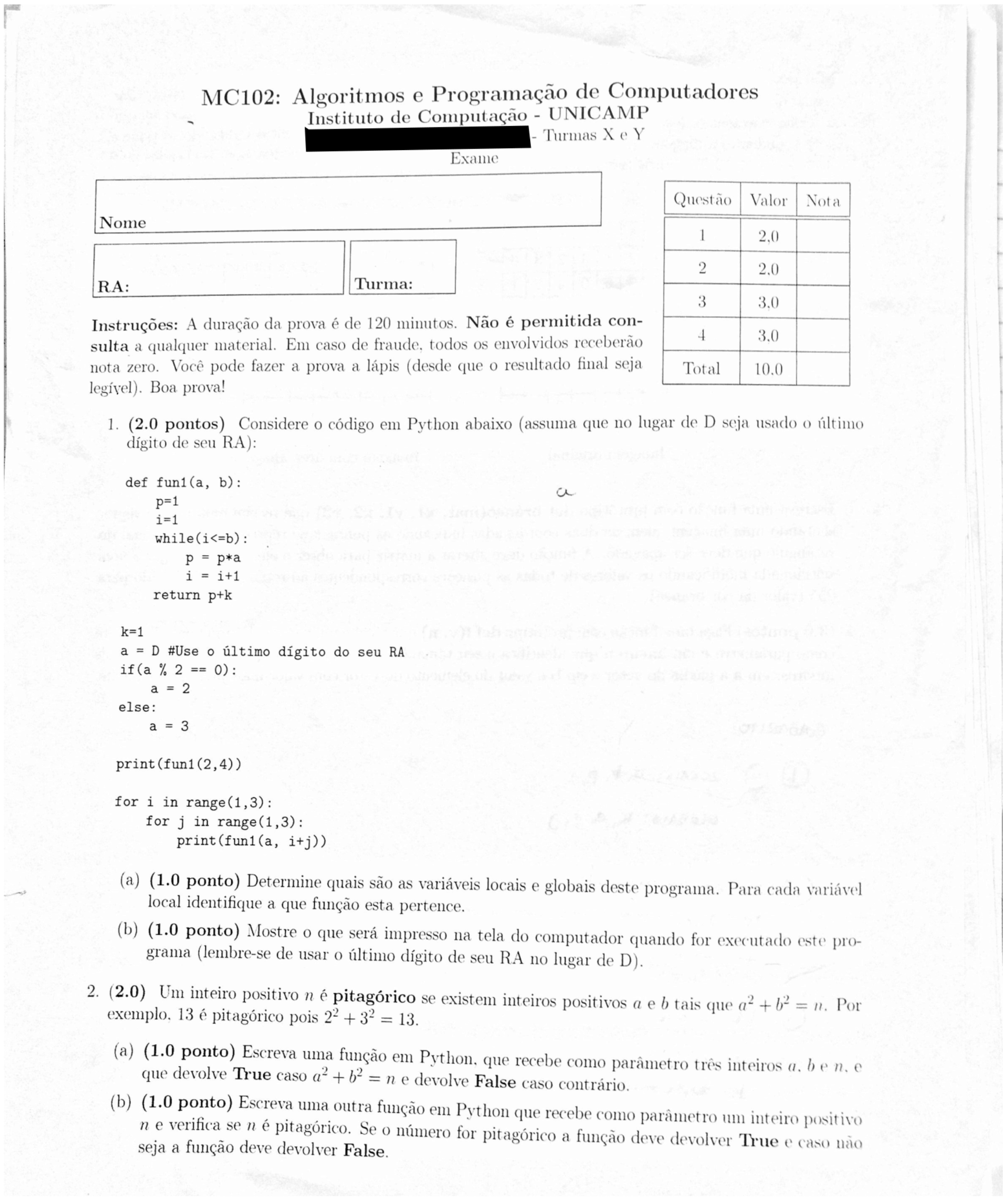
   Por exemplo, para a matriz abaixo seu programa deveria retornar **True**.

```
00000000
0ab0ca00
00c000c0
00000000
```

4. (**3.0**) Escreva uma função que recebe um inteiro **n** passado por parâmetro e retorna uma tupla **(a,b)** onde **a** é o maior número primo que é menor ou igual a n e **b** é o menor número primo que é maior ou igual a n. Por exemplo se $n = 10$ devemos ter $a = 7$ e $b = 11$.

Figure 4: Final Exam, MC102 Section 2/2015, I1

A.3 MC102, 2nd Term 2015, Instructor I1, Retake Exam



MC102: Algoritmos e Programação de Computadores

Instituto de Computação - UNICAMP

[redacted] - Turmas X e Y

Exame

Nome	
RA:	Turma:

Questão	Valor	Nota
1	2,0	
2	2,0	
3	3,0	
4	3,0	
Total	10,0	

Instruções: A duração da prova é de 120 minutos. **Não é permitida consulta** a qualquer material. Em caso de fraude, todos os envolvidos receberão nota zero. Você pode fazer a prova a lápis (desde que o resultado final seja legível). Boa prova!

1. **(2.0 pontos)** Considere o código em Python abaixo (assuma que no lugar de D seja usado o último dígito de seu RA):

```
def fun1(a, b):
    p=1
    i=1
    while(i<=b):
        p = p*a
        i = i+1
    return p+k

k=1
a = D #Use o último dígito do seu RA
if(a % 2 == 0):
    a = 2
else:
    a = 3

print(fun1(2,4))

for i in range(1,3):
    for j in range(1,3):
        print(fun1(a, i+j))
```

   (a) **(1.0 ponto)** Determine quais são as variáveis locais e globais deste programa. Para cada variável local identifique a que função esta pertence.

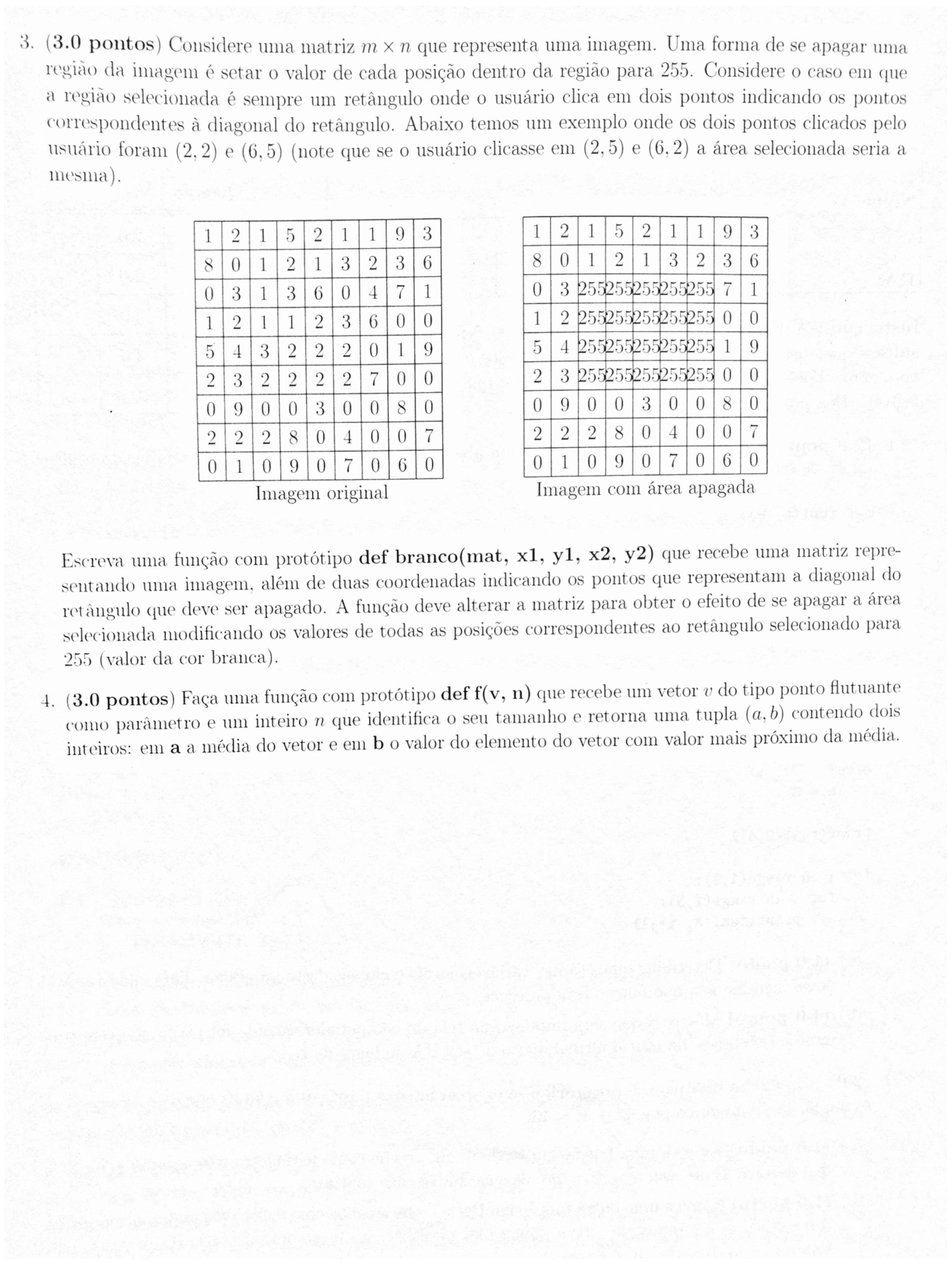
   (b) **(1.0 ponto)** Mostre o que será impresso na tela do computador quando for executado este programa (lembre-se de usar o último dígito de seu RA no lugar de D).

2. **(2.0)** Um inteiro positivo n é **pitagórico** se existem inteiros positivos a e b tais que $a^2 + b^2 = n$. Por exemplo, 13 é pitagórico pois $2^2 + 3^2 = 13$.

   (a) **(1.0 ponto)** Escreva uma função em Python, que recebe como parâmetro três inteiros a, b e n, e que devolve **True** caso $a^2 + b^2 = n$ e devolve **False** caso contrário.

   (b) **(1.0 ponto)** Escreva uma outra função em Python que recebe como parâmetro um inteiro positivo n e verifica se n é pitagórico. Se o número for pitagórico a função deve devolver **True** e caso não seja a função deve devolver **False**.

Figure 5: Retake Exam, MC102 Section 2/2015, I1



3. (**3.0 pontos**) Considere uma matriz $m \times n$ que representa uma imagem. Uma forma de se apagar uma região da imagem é setar o valor de cada posição dentro da região para 255. Considere o caso em que a região selecionada é sempre um retângulo onde o usuário clica em dois pontos indicando os pontos correspondentes à diagonal do retângulo. Abaixo temos um exemplo onde os dois pontos clicados pelo usuário foram $(2, 2)$ e $(6, 5)$ (note que se o usuário clicasse em $(2, 5)$ e $(6, 2)$ a área selecionada seria a mesma).

1	2	1	5	2	1	1	9	3
8	0	1	2	1	3	2	3	6
0	3	1	3	6	0	4	7	1
1	2	1	1	2	3	6	0	0
5	4	3	2	2	2	0	1	9
2	3	2	2	2	2	7	0	0
0	9	0	0	3	0	0	8	0
2	2	2	8	0	4	0	0	7
0	1	0	9	0	7	0	6	0

Imagem original

1	2	1	5	2	1	1	9	3
8	0	1	2	1	3	2	3	6
0	3	255	255	255	255	255	7	1
1	2	255	255	255	255	255	0	0
5	4	255	255	255	255	255	1	9
2	3	255	255	255	255	255	0	0
0	9	0	0	3	0	0	8	0
2	2	2	8	0	4	0	0	7
0	1	0	9	0	7	0	6	0

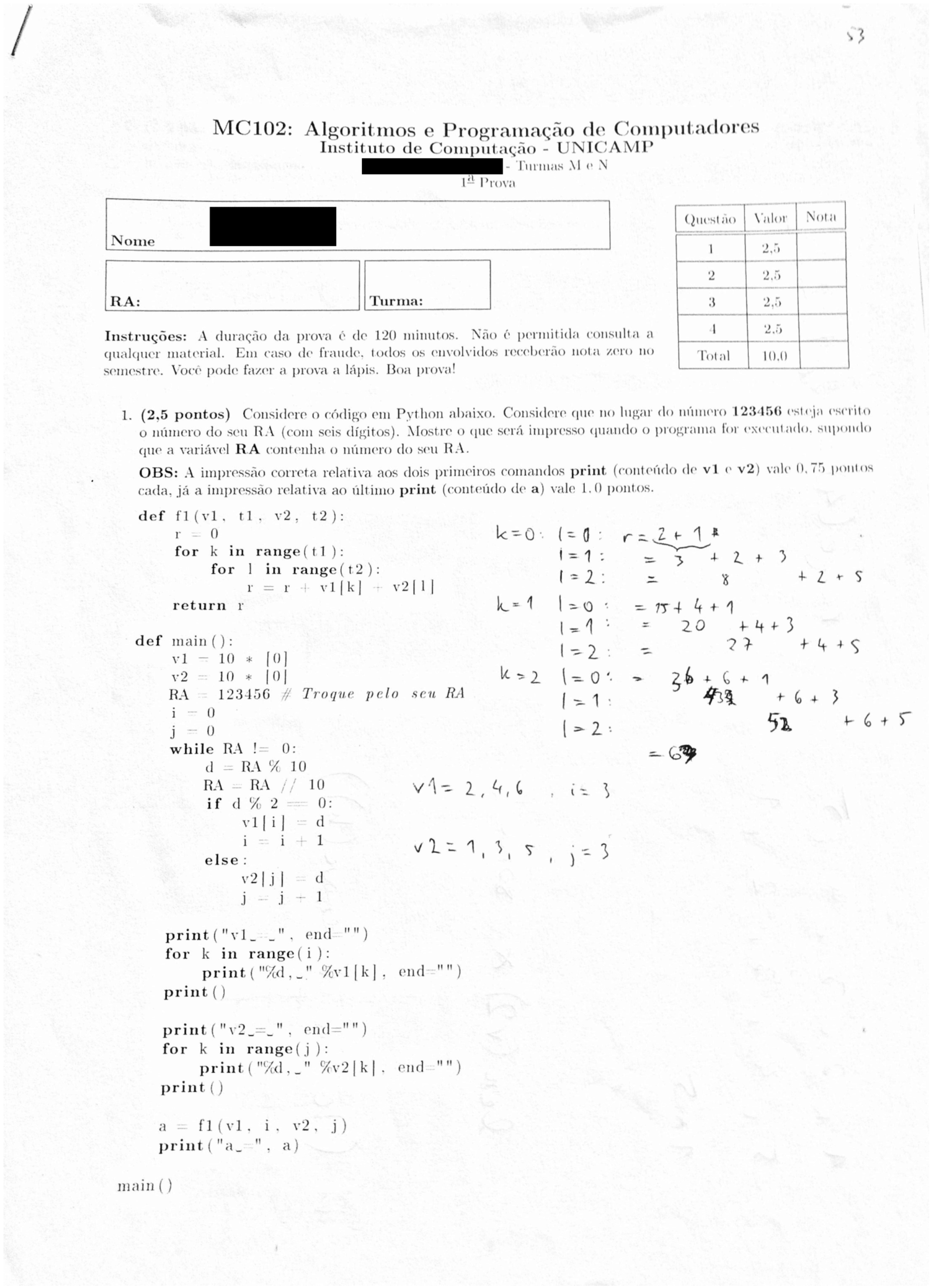
Imagem com área apagada

Escreva uma função com protótipo **def branco(mat, x1, y1, x2, y2)** que recebe uma matriz representando uma imagem, além de duas coordenadas indicando os pontos que representam a diagonal do retângulo que deve ser apagado. A função deve alterar a matriz para obter o efeito de se apagar a área selecionada modificando os valores de todas as posições correspondentes ao retângulo selecionado para 255 (valor da cor branca).

4. (**3.0 pontos**) Faça uma função com protótipo **def f(v, n)** que recebe um vetor v do tipo ponto flutuante como parâmetro e um inteiro n que identifica o seu tamanho e retorna uma tupla (a, b) contendo dois inteiros: em **a** a média do vetor e em **b** o valor do elemento do vetor com valor mais próximo da média.

Figure 6: Retake Exam, MC102 Section 2/2015, I1, page 2

A.4 MC102, 2nd Term 2016, Instructor I2, Midterm Exam



/

53

MC102: Algoritmos e Programação de Computadores
Instituto de Computação - UNICAMP
[redacted] - Turmas M e N
1ª Prova

Nome [redacted]	
RA:	Turma:

Questão	Valor	Nota
1	2,5	
2	2,5	
3	2,5	
4	2,5	
Total	10,0	

Instruções: A duração da prova é de 120 minutos. Não é permitida consulta a qualquer material. Em caso de fraude, todos os envolvidos receberão nota zero no semestre. Você pode fazer a prova a lápis. Boa prova!

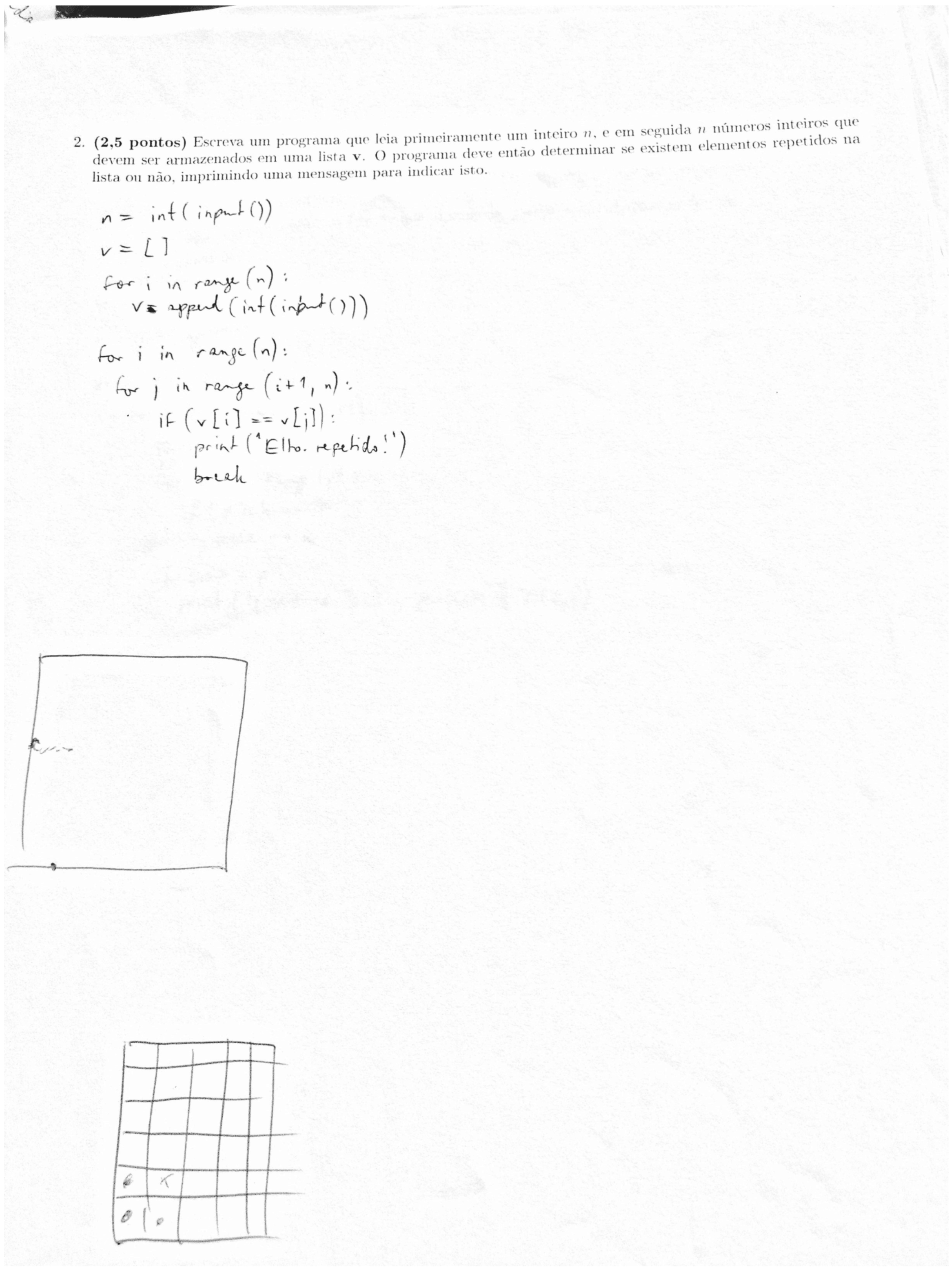
1. **(2,5 pontos)** Considere o código em Python abaixo. Considere que no lugar do número **123456** esteja escrito o número do seu RA (com seis dígitos). Mostre o que será impresso quando o programa for executado, supondo que a variável **RA** contenha o número do seu RA.

   OBS: A impressão correta relativa aos dois primeiros comandos **print** (conteúdo de **v1** e **v2**) vale 0,75 pontos cada, já a impressão relativa ao último **print** (conteúdo de **a**) vale 1.0 pontos.

```
def f1(v1, t1, v2, t2):
    r = 0
    for k in range(t1):
        for l in range(t2):
            r = r + v1[k] + v2[l]
    return r

def main():
    v1 = 10 * [0]
    v2 = 10 * [0]
    RA = 123456 # Troque pelo seu RA
    i = 0
    j = 0
    while RA != 0:
        d = RA % 10
        RA = RA // 10
        if d % 2 == 0:
            v1[i] = d
            i = i + 1
        else:
            v2[j] = d
            j = j + 1

    print("v1 = ", end="")
    for k in range(i):
        print("%d, " %v1[k], end="")
    print()

    print("v2 = ", end="")
    for k in range(j):
        print("%d, " %v2[k], end="")
    print()

    a = f1(v1, i, v2, j)
    print("a = ", a)

main()
```

```
k=0: l=0:  r = 2 + 1
     l=1:  = 3 + 2 + 3
     l=2:  =     8       + 2 + 5
k=1  l=0:  = 15 + 4 + 1
     l=1:  =    20    + 4 + 3
     l=2:  =        27     + 4 + 5
k=2  l=0:  =  36 + 6 + 1
     l=1:       43       + 6 + 3
     l=2:           52        + 6 + 5

           = 6[illegible]

v1 = 2, 4, 6  , i = 3

v2 = 1, 3, 5  , j = 3
```

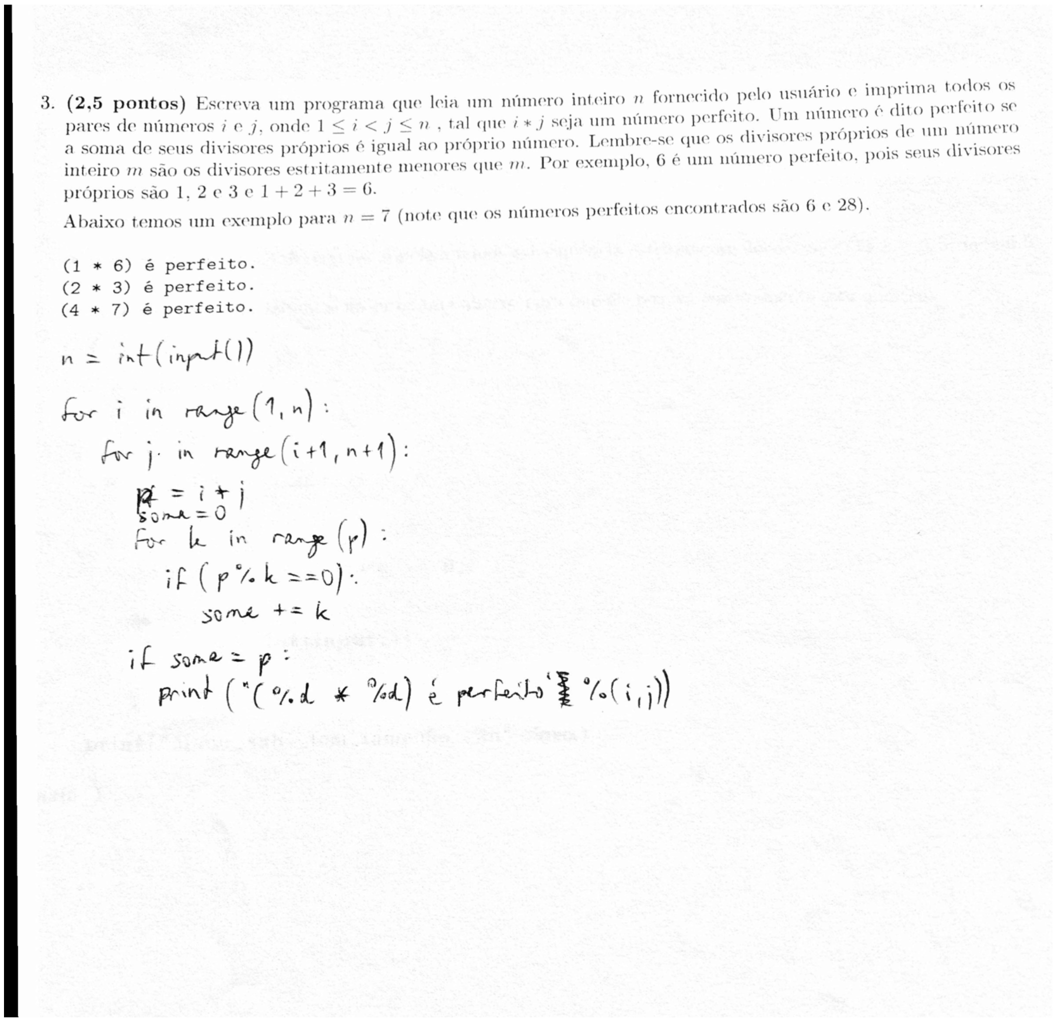
Figure 7: Midterm Exam, MC102 Section 2/2016, I2, page 1.



2. **(2,5 pontos)** Escreva um programa que leia primeiramente um inteiro n, e em seguida n números inteiros que devem ser armazenados em uma lista **v**. O programa deve então determinar se existem elementos repetidos na lista ou não, imprimindo uma mensagem para indicar isto.

```
n = int(input())
v = [ ]
for i in range(n):
   v.append(int(input()))

for i in range(n):
  for j in range(i+1, n):
      if (v[i] == v[j]):
          print("Elts. repetidos!")
          break
```

Figure 8: Midterm Exam, MC102 Section 2/2016, I2, page 2



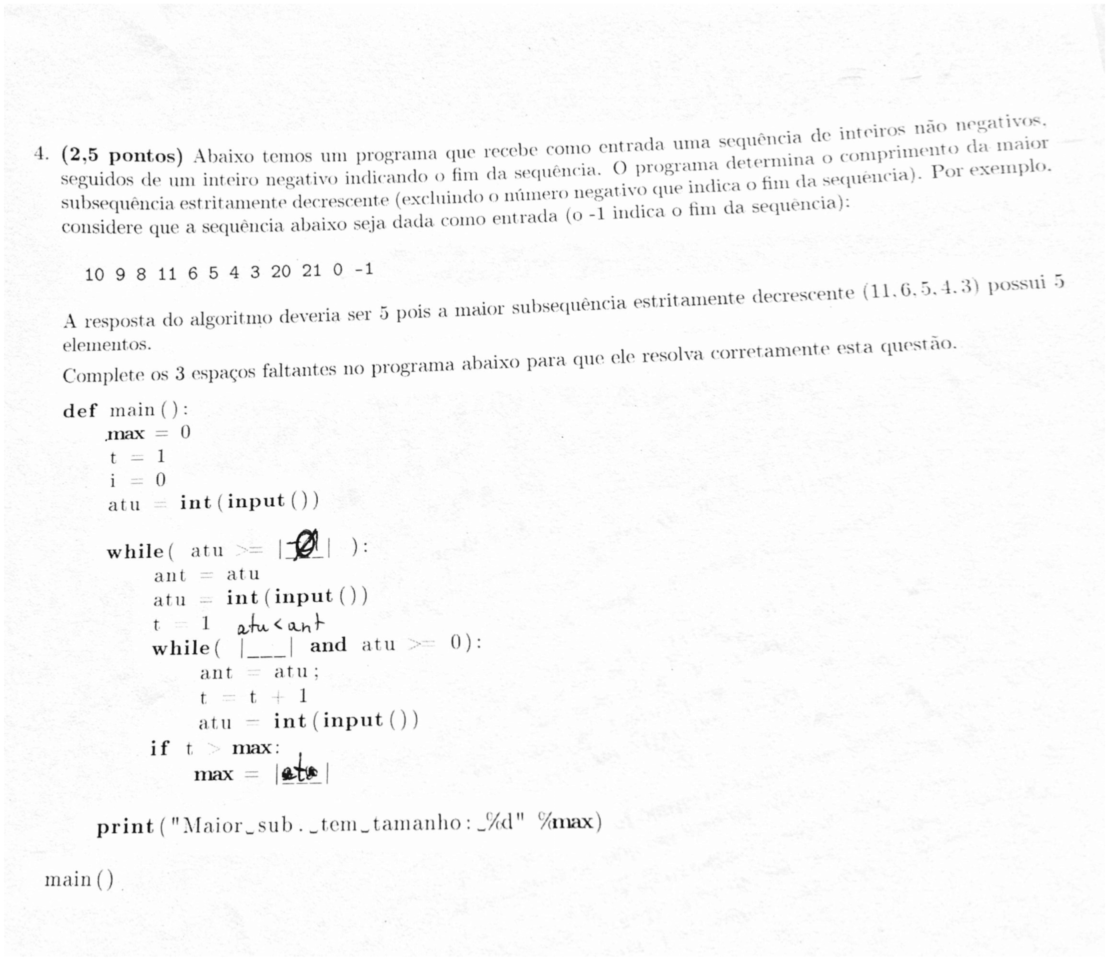
3. **(2,5 pontos)** Escreva um programa que leia um número inteiro n fornecido pelo usuário e imprima todos os pares de números i e j, onde $1 \leq i < j \leq n$, tal que $i * j$ seja um número perfeito. Um número é dito perfeito se a soma de seus divisores próprios é igual ao próprio número. Lembre-se que os divisores próprios de um número inteiro m são os divisores estritamente menores que m. Por exemplo, 6 é um número perfeito, pois seus divisores próprios são 1, 2 e 3 e $1 + 2 + 3 = 6$.

Abaixo temos um exemplo para $n = 7$ (note que os números perfeitos encontrados são 6 e 28).

```
(1 * 6) é perfeito.
(2 * 3) é perfeito.
(4 * 7) é perfeito.
```

```
n = int(input())

for i in range(1, n):
    for j in range(i+1, n+1):
        p = i * j
        soma = 0
        for k in range(p):
            if (p % k == 0):
                soma += k

        if soma = p:
            print("(%d * %d) é perfeito" % (i,j))
```

Figure 9: Midterm Exam, MC102 Section 2/2016, I2, page 3



4. **(2,5 pontos)** Abaixo temos um programa que recebe como entrada uma sequência de inteiros não negativos, seguidos de um inteiro negativo indicando o fim da sequência. O programa determina o comprimento da maior subsequência estritamente decrescente (excluindo o número negativo que indica o fim da sequência). Por exemplo, considere que a sequência abaixo seja dada como entrada (o -1 indica o fim da sequência):

```
10 9 8 11 6 5 4 3 20 21 0 -1
```

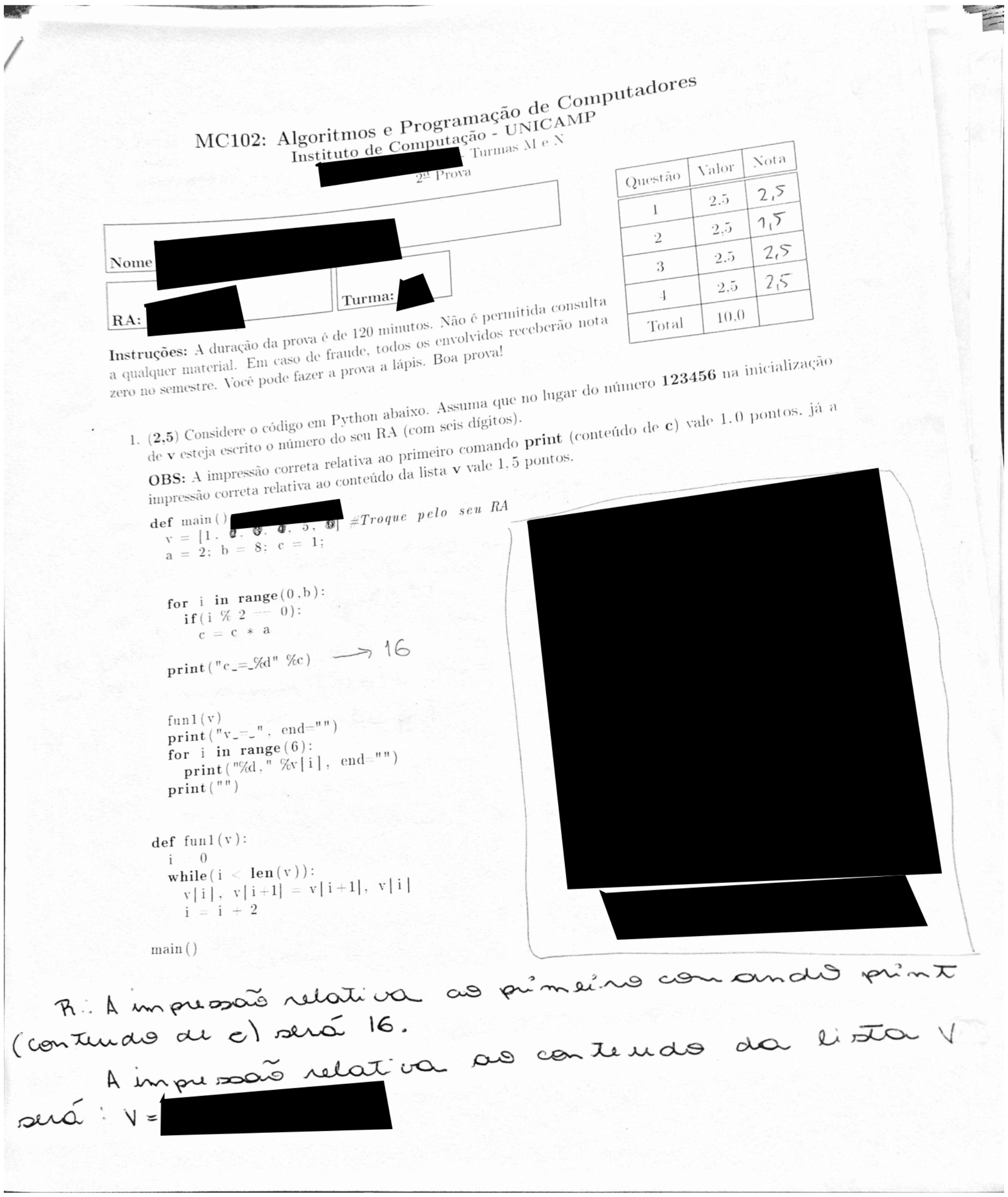
A resposta do algoritmo deveria ser 5 pois a maior subsequência estritamente decrescente (11, 6, 5, 4, 3) possui 5 elementos.

Complete os 3 espaços faltantes no programa abaixo para que ele resolva corretamente esta questão.

```
def main():
    max = 0
    t = 1
    i = 0
    atu = int(input())

    while( atu >= |_0_| ):
        ant = atu
        atu = int(input())
        t = 1    atu<ant
        while( |___| and atu >= 0):
            ant = atu;
            t = t + 1
            atu = int(input())
        if t > max:
            max = |_t_|

    print("Maior␣sub.␣tem␣tamanho:␣%d" %max)

main()
```

Figure 10: Midterm Exam, MC102 Section 2/2016, I2, page 4

A.5 MC102, 2nd Term 2016, Instructor I2, Final Exam



MC102: Algoritmos e Programação de Computadores
Instituto de Computação - UNICAMP
[redacted] - Turmas M e N
2ª Prova

Nome [redacted]

RA: [redacted] Turma: [redacted]

Questão	Valor	Nota
1	2.5	2,5
2	2.5	1,5
3	2.5	2,5
4	2.5	2,5
Total	10.0	

Instruções: A duração da prova é de 120 minutos. Não é permitida consulta a qualquer material. Em caso de fraude, todos os envolvidos receberão nota zero no semestre. Você pode fazer a prova a lápis. Boa prova!

1. (**2,5**) Considere o código em Python abaixo. Assuma que no lugar do número **123456** na inicialização de **v** esteja escrito o número do seu RA (com seis dígitos).

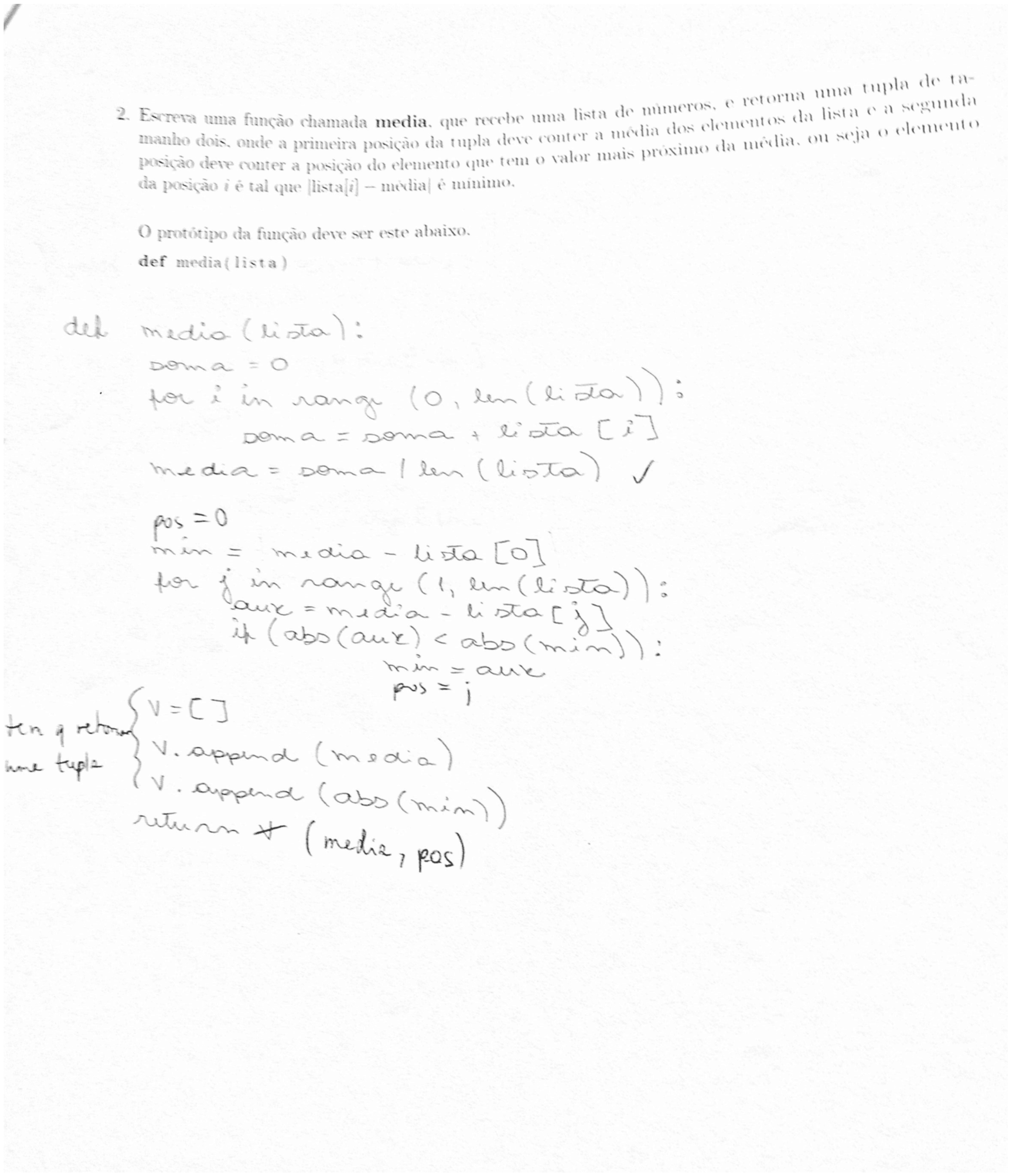
   OBS: A impressão correta relativa ao primeiro comando **print** (conteúdo de **c**) vale 1.0 pontos, já a impressão correta relativa ao conteúdo da lista **v** vale 1.5 pontos.

```
def main() [redacted] #Troque pelo seu RA
  v = [1, [redacted] 5, [redacted]]
  a = 2; b = 8; c = 1;

  for i in range(0,b):
    if(i % 2 == 0):
      c = c * a

  print("c = %d" %c)   → 16

  fun1(v)
  print("v = ", end="")
  for i in range(6):
    print("%d, " %v[i], end="")
  print("")

def fun1(v):
  i = 0
  while(i < len(v)):
    v[i], v[i+1] = v[i+1], v[i]
    i = i + 2

main()
```

[redacted]

R.: A impressão relativa ao primeiro comando print (conteúdo de c) será 16.
A impressão relativa ao conteúdo da lista V será: V = [redacted]

Figure 11: Final Exam, MC102 Section 2/2016, I2. This particular copy is a student's submission, and therefore all identifying information has been removed.



2. Escreva uma função chamada **media**, que recebe uma lista de números, e retorna uma tupla de tamanho dois, onde a primeira posição da tupla deve conter a média dos elementos da lista e a segunda posição deve conter a posição do elemento que tem o valor mais próximo da média, ou seja o elemento da posição *i* é tal que |lista[*i*] − média| é mínimo.

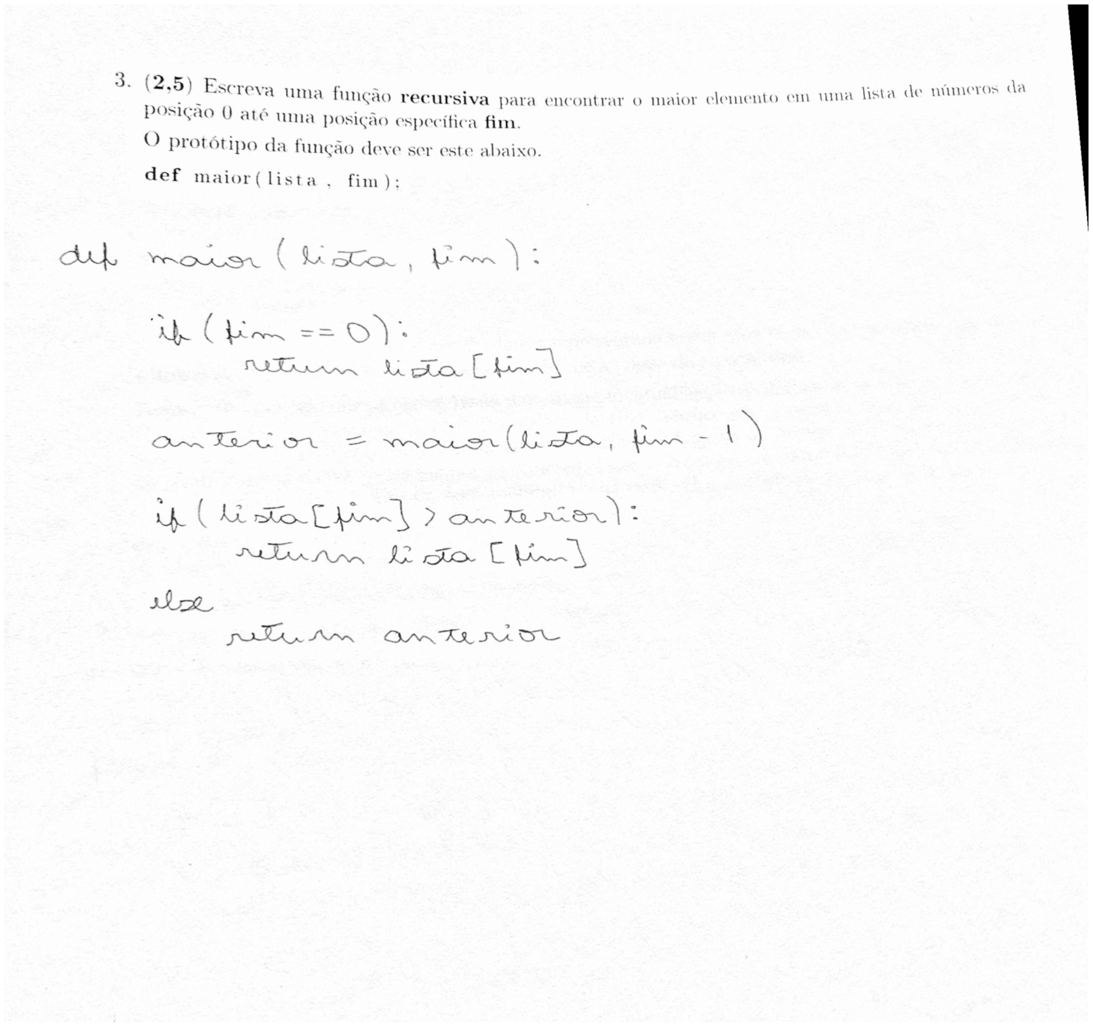
O protótipo da função deve ser este abaixo.

def media(lista)

```
def media(lista):
    soma = 0
    for i in range(0, len(lista)):
        soma = soma + lista[i]
    media = soma / len(lista)  ✓

    pos = 0
    min = media - lista[0]
    for j in range(1, len(lista)):
        aux = media - lista[j]
        if (abs(aux) < abs(min)):
            min = aux
            pos = j
    V = []
    V.append(media)
    V.append(abs(min))
    return ~~v~~ (media, pos)
```

tem q retorna uma tupla

Figure 12: Final Exam, MC102 Section 2/2016, I2, page 2

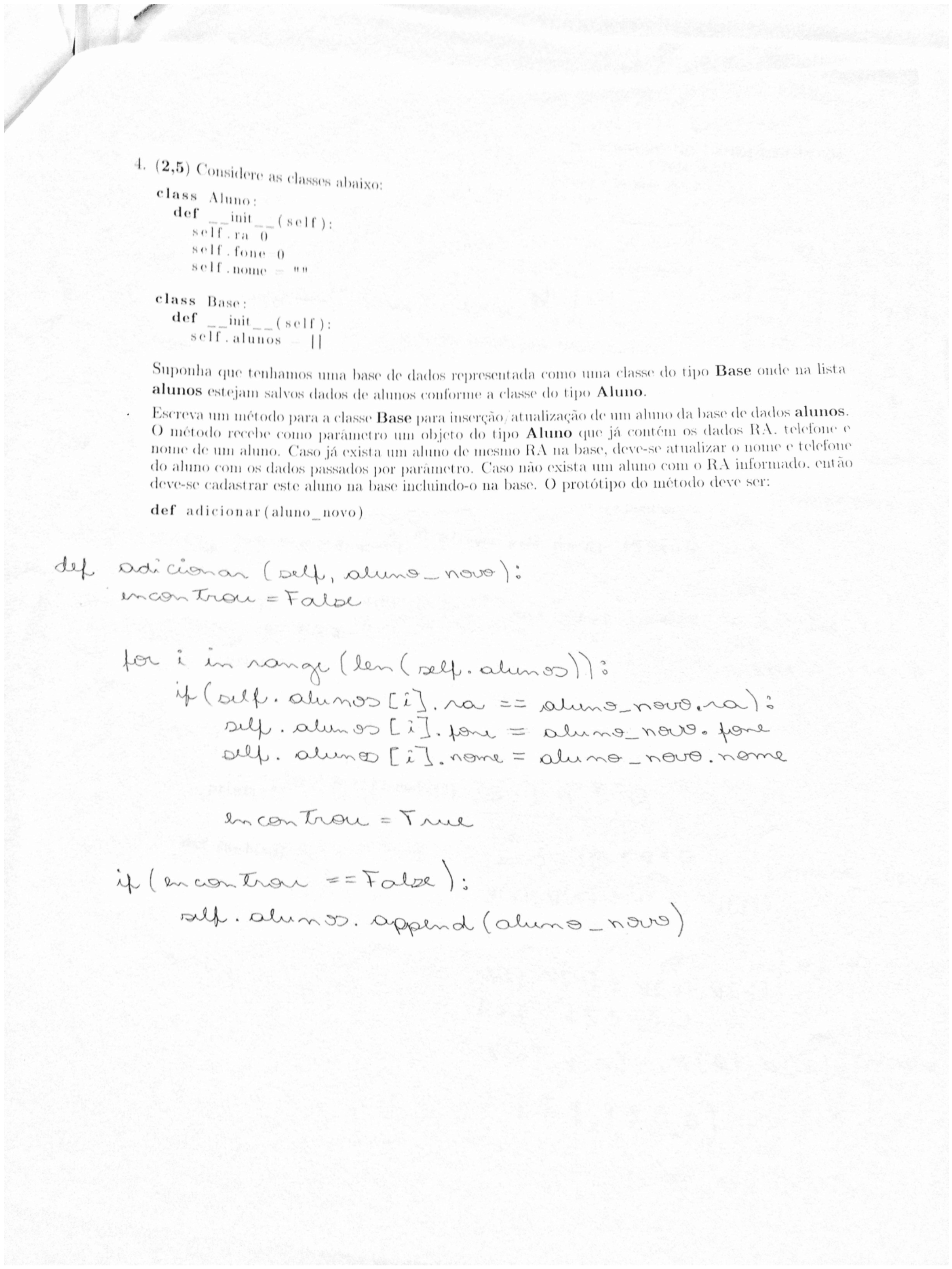


3. (2,5) Escreva uma função **recursiva** para encontrar o maior elemento em uma lista de números da posição 0 até uma posição específica **fim**.
O protótipo da função deve ser este abaixo.

```
def maior(lista , fim):
```

```
def maior(lista, fim):

    if (fim == 0):
        return lista[fim]

    anterior = maior(lista, fim - 1)

    if (lista[fim] > anterior):
        return lista[fim]
    else
        return anterior
```

Figure 13: Final Exam, MC102 Section 2/2016, I2, page 3



4. **(2,5)** Considere as classes abaixo:

```
class Aluno:
  def __init__(self):
    self.ra = 0
    self.fone = 0
    self.nome = ""

class Base:
  def __init__(self):
    self.alunos = []
```

Suponha que tenhamos uma base de dados representada como uma classe do tipo **Base** onde na lista **alunos** estejam salvos dados de alunos conforme a classe do tipo **Aluno**.

- Escreva um método para a classe **Base** para inserção/atualização de um aluno da base de dados **alunos**. O método recebe como parâmetro um objeto do tipo **Aluno** que já contém os dados RA, telefone e nome de um aluno. Caso já exista um aluno de mesmo RA na base, deve-se atualizar o nome e telefone do aluno com os dados passados por parâmetro. Caso não exista um aluno com o RA informado, então deve-se cadastrar este aluno na base incluindo-o na base. O protótipo do método deve ser:

```
def adicionar(aluno_novo)
```

```
def adicionar (self, aluno_novo):
    encontrou = False

    for i in range(len(self.alunos)):
        if(self.alunos[i].ra == aluno_novo.ra):
            self.alunos[i].fone = aluno_novo.fone
            self.alunos[i].nome = aluno_novo.nome

            encontrou = True

    if(encontrou == False):
        self.alunos.append(aluno_novo)
```

Figure 14: Final Exam, MC102 Section 2/2016, I2, page 4

Appendix B Translations of Exams into English

This appendix contains the translations into English of the exams shown in Appendix A, following the same numbering of subsections.

This translation was done with the intent of providing insight into the students' experiences taking these exams. To that end, all variable and function names in Portuguese have been translated into English as well.

B.1 Midterm Exam, MC102 2nd Term 2015

MC102: Algorithms and Computer Programming

Institute of Computing – UNICAMP

[Instructor's name omitted] – Classes X and Y

1st Exam

Name:

Student ID:

Instructions: The length of the exam is 120 minutes. **You may not consult** any material. In the event of fraud, all of those involved will be given grade zero. You may take this exam in pencil (provided that the final result is legible). Have a good test!

1. (2.0 points) Consider the Python code below (substitute D with the last digit of your student ID):

```
def f1(n, k):
    a=1
    b=1
    for i in range(1,n+1):
        aux = b
        b = b+a
        a = aux
    return b+k+j

j=1

x=D # D is the last digit of your student ID

print(f1(5,0))

for i in range(3):
    for j in range(3):
        if (i+j)%2 == 0):
            print(f1(i+j,x))
```

(a) **(0.5 points)** Determine the local and global variables of this program. For each local variable, identify the function to which it belongs.

(b) **(1.5 points)** Show what will be printed on the computer screen when this program is run (remember to replace D with the last digit of your student ID).

2. (2.0 points) Write a function that receives a list of integers as a parameter, and another integer value V. The function must return True if there exist two numbers in distinct positions of the list whose sum equals V, and False if not. The prototype of the function must be:
def func(arr, V)

3. (3.0 points) Write a program that reads an integer *n* supplied by the user and prints out a 'square' of *n* rows and *n* columns, where, at row *i*, column *j*, the value 1 is printed in case *i* and *j* are coprime, and 0 if they are not. Two numbers *a* and *b* are coprime if there exists no divisor *d* > *1* that is common to both. For instance, 15 and 8 are coprime, because the divisors of 8, which are 2, 4, and 8, are not divisors of 15. Below, we have an example for *n = 9*.

```
  1 2 3 4 5 6 7 8 9
1 1 1 1 1 1 1 1 1 1
2 1 0 1 0 1 0 1 0 1
3 1 1 0 1 1 0 1 1 0
4 1 0 1 0 1 0 1 0 1
5 1 1 1 1 0 1 1 1 1
6 1 0 0 0 1 0 1 0 0
7 1 1 1 1 1 1 0 1 1
8 1 0 1 0 1 0 1 0 1
9 1 1 0 1 1 0 1 1 0
```

The numbers from 1 to 9 in the first column and first row were included above merely as an illustration. The output of your program for *n = 9* must be simply:

```
1 1 1 1 1 1 1 1 1
1 0 1 0 1 0 1 0 1
1 1 0 1 1 0 1 1 0
1 0 1 0 1 0 1 0 1
1 1 1 1 0 1 1 1 1
1 0 0 0 1 0 1 0 0
1 1 1 1 1 1 0 1 1
1 0 1 0 1 0 1 0 1
1 1 0 1 1 0 1 1 0
```

Hint: Remember that, to print a value *a* without automatically beginning a new line, you must enter

```
print(a,end="")
```

- Write and implement a function **def coprimes(a, b)** that returns True if two numbers **a** and **b** passed as parameters are coprime, and returns False otherwise. Then use that function as part of the solution for this question.

4. (3.0 Points) Here is code in Python that implements the **BubbleSort** algorithm.

```
def bubbleSort (arr):
    for i in range(len(arr)-1,0,-1):
          for j in range(i):
                if arr[j] > arr[j+1]:
                      arr[j], arr[j+1] = arr[j+1], arr[j]
```

(a) Consider two positive integers *a* and *b*. Let us define a rule to determine whether a number *a* is greater than some number *b* in the following manner: even numbers are greater than prime numbers. If both numbers are even, the one that is coprime of 42 will be the greater of the two. If there is still a tie, the one with the greater value is greater. If both numbers are odd, the one that is coprime of 15 will be the greater of the two. If there is still a tie, the number with greater value will be greater.
Obs.: You may presume the existence of the function **def coprimes(a, b)** that returns True if two numbers **a** and **b** passed as parameters are coprime, and returns False otherwise.

(b) Create a function **def greater(a, b)** that returns True if *a* is greater than *b* by the rule above, and returns False otherwise.

(c) Re-implement the BubbleSort algorithm so that it may sort an array of positive integers, but adopting the rule established in (a) to determine whether a number is greater than another.

B.2 Final Exam, MC102 2nd Term 2015

MC102: Algorithms and Computer Programming

Institute of Computing – UNICAMP

[Instructor's name omitted] – Classes X and Y

2nd Exam

Name:

Student ID:

Instructions: In the event of fraud, all of those involved will be given grade zero. You may take this exam in pencil.

1. (**2.0**) Write a function **def space_text(s1)** that receives a string **s1** by parameter and returns another string as a result of including one extra space character for every space character previously existing in **s1**.

Examples:

```
cafe forte do Brasil  --->
cafe  forte  do  Brasil
ask not what your country can do for you, ask what you can do for your
country. -->
ask  not  what  your  country  can  do  for  you,  ask  what  you  can  do
for  your  country.
```

Hint: Remember that **split(s)** generates a list where each element is a character from **s** and the method **''.join(l)** receives a list **l** of characters and turns it into a string.

2. (**2.0**) Write a **recursive** function to find the largest element of a list of integers of length **n**. The function must return the largest number. The prototype of the function is **def greater(v, n)**, where *n* represents the number of elements of the list **v**.

3. (**3.0**) In a classroom, desks are arranged in *m* rows and *n* columns. An instructor will apply an exam for MC102 in three different versions. Each desk will be represented by the characters `'a'`, `'b'`, or `'c'`, corresponding to the versions of the exam, or a character zero (`'0'`), if it is empty. On applying an exam, he has established the following rules:

(a) no one is to sit in the first and last columns, nor in the first and last rows.
(b) for any one student taking the test, the 8 desks surrounding that student must either be empty or must have a student taking a test of a different version.

Write a function that receives as parameters: a matrix of characters (a list of lists of characters), the dimensions *m* and *n* of that matrix, where the matrix represents a possible allocation of exams in that classroom.
The function must return **True** if the distribution follows the instructor's rules, and **False** otherwise.

The prototype is: **def seating_arrangement(classroom, m, n)**.

For instance, for the matrix below your program should return **True**.

```
00000000
0ab0ca00
00c000c0
00000000
```

4. (**3.0**) Write a function that receives an integer **n** passed as a parameter and returns a tuple **(a,b)**, where **a** is the greatest prime number that is less than or equal to *n* and **b** is the smallest prime number that is greater than or equal to *n*. For instance, if *n = 10*, then *a = 7* and *b = 11*.

B.3 Retake Exam, MC102 2nd Term 2015

MC102: Algorithms and Computer Programming

Institute of Computing – UNICAMP

[Instructor’s name omitted] – Classes X and Y

Retake Exam

Name:

Student ID:

Instructions: The length of the exam is 120 minutes. You may not consult any material. In the event of fraud, all of those involved will be given grade zero. You may take this exam in pencil (provided that the final result is legible). Have a good test!

1. **(2.0 points)** Consider the Python code below (assume that D is replaced by the last digit of your Student ID):

```
def fun1(a, b):
      p=1
      i=1
      while(i<=b):
            p = p*A
            i = i+1
      return p+k

k=1
a = D #Use the last digit of your student ID
if(a % 2 == 0):
      a = 2
else:
      a = 3

print(fun1(2,4))

for i in range(1,3):
      for j in range(1,3):
            print(fun1(a, i+j))
```

(a) **(1.0 point)** Determine which of the variables in this program are local and which are global. For each local variable, identify the function to which it belongs.

(b) **(1.0 point)** Show what will be printed out on the computer screen when this program is executed (remember to use the last digit of your student ID instead of D).

2. **(2.0)** A positive number n is **Pythagorean** if there exist positive integers a and b such that $a^2 + b^2 = n$. For instance, 13 is Pythagorean, since $2^2 + 3^2 = 13$.

(a) **(1.0 point)** Write a function in Python that receives three integers a, b, n as parameters and returns **True** if $a^2 + b^2 = n$ and returns **False** otherwise.

(b) **(1.0 point)** Write another function in Python that receives a positive integer n as a parameter and verifies in n is Pythagorean. If the number is Pythagorean the function should return **True**, and, if not, the function should return **False.**

3. **(3.0 points)** Consider an $m \times n$ matrix representing an image. One way of erasing a region of that image is setting the value of each position inside the region to 255. Consider the case where the selected region is always a rectangle, and the user indicates that rectangle by clicking on two points corresponding to the rectangle's diagonal. Below we have an example where the two points clicked by the user were (2, 2) and (6, 5) (note that if the user had clicked on (2, 5) and (6, 2) the selected area would have been the same).

1	2	1	5	2	1	1	9	3
8	0	1	2	1	3	2	3	6
0	3	1	3	6	0	4	7	1
1	2	1	1	2	3	6	0	0
5	4	3	2	2	2	0	1	9
2	3	2	2	2	2	7	0	0
0	9	0	0	3	0	0	8	0
2	2	2	8	0	4	0	0	7
0	1	0	9	0	7	0	6	0

Original image

1	2	1	5	2	1	1	9	3
8	0	1	2	1	3	2	3	6
0	3	255	255	255	255	255	7	1
1	2	255	255	255	255	255	0	0
5	4	255	255	255	255	255	1	9
2	3	255	255	255	255	255	0	0
0	9	0	0	3	0	0	8	0
2	2	2	8	0	4	0	0	7
0	1	0	9	0	7	0	6	0

Image with erased region

Write a function with the prototype **def white(mat, x1, y1, x2, y2)** which receives a matrix representing an image, as well as two coordinates indicating the points which represent the diagonal of the rectangle to be erased. The function should alter the matrix so as to obtain the effect of erasing the selected area by modifying the values of all positions corresponding to the selected rectangle to 255 (value of the color white).

4. **(3.0 points)** Write a function with the prototype **def f(v, n)** which receives an array *v* of floating-point values as a parameter and an integer *n* identifying its length, and returns a tuple *(a, b)* containing two integers: in **a**, the mean of the array, and in **b** the value of the vector of the array whose value is closest to the mean.

B.4 Midterm Exam, MC102 2nd Term 2016

MC102: Algorithms and Computer Programming

Institute of Computing – UNICAMP

[Instructor's name omitted] – Classes M and N

1st Exam

Name:

Student ID:

Instructions: The length of the exam is 120 minutes. You may not consult any material. In the event of fraud, all of those involved will be given grade zero for the entire term. You may take this exam in pencil. Have a good test!

1. **(2.5 points)** Consider the Python code below. Consider that, instead of the number **123456**, your six-digit student ID is written. Show what will be printed out when the program is executed, assuming the variable **RA** contains your student ID.

OBS: The correct printout regarding the first two **print** statements (the content of **v1** and **v2**) is worth 0.75 points, whereas the printout regarding the last **print** (the content of **a**) is worth 1.0 point.

```
def f1(v1, t2, v2, t2):
      r = 0
      for k in range(t1):
            for l in range(t2):
                  r = r + v1[k] + v2[l]
      return r

def main():
      v1 = 10 * [0]
      v2 = 10 * [0]
      RA = 123456 # Replace with your student ID
      i = 0
      j = 0
      while RA != 0:
            d = RA % 10
            RA = RA // 10
            if d % 2 == 0:
                  v1[i] = d
                  i = i + 1
            else:
                  v2[j] = d
                  j = j + 1

print("v1␣=␣", end = "")
for k in range(i):
      print("%d,␣" %v1[k], end="")
print()

print("v2␣=␣", end="")
for k in range(j):
      print("%d, ␣" %v2[k], end="")
print()

a = f1(v1, i, v2, j)
print("a␣=", a)

main()
```

2. **(2.5 points)** Write a program which first reads an integer *n*, then *n* integers that are stored in a list **v**. The program should then determine whether there are repeated elements in the list, printing out a message to indicate that.

3. **(2.5 points)** Write a program that reads an integer *n* supplied by the user and prints out all pairs of number *i* and *j*, where $1 \leq i < j \leq n$, such that $i * j$ is a perfect number. A number is said to be perfect if the sum of its proper divisors equals the number itself. Recall that the proper divisors of an integer *m* are all divisors strictly less than *m*. For instance, 6 is a perfect number, as its proper divisors are 1, 2, and 3, and $1 + 2 + 3 = 6$.

Below we have an example for $n = 7$ (note that the perfect numbers found are 6 and 28).

```
(1 * 6) is perfect.
(2 * 3) is perfect.
(4 * 7) is perfect.
```

4. **(2.5 points)** Below we have a program that receives as input a sequence of non-negative integers, followed by a negative integer indicating the end of the sequence. The program determines the length of the longest subsequence that is strictly decreasing (excluded the negative number that indicates its end). For instance, consider that the sequence below is given as an input (the -1 indicates the end of the sequence):

```
10 9 8 11 6 5 4 3 20 21 0 -1
```

The algorithm should return 5, as the longest strictly decreasing subsequence (11, 6, 5, 4, 3) contains 5 elements.

Complete the 3 blank spaces in the program below so that it correctly solves this question.

```
def main():
      max = 0
      t = 1
      i = 0
      curr = int(input())

      while( curr >= _________):
            prev = curr
            curr = int(input())
            t = 1
            while(_________ and curr >= 0):
                  prev = curr
                  t = t + 1
                  curr = int(input())
            if t > max:
                  max = _________

      print("Greatest␣sub. ␣has␣size␣%d" % max)

main()
```

B.5 Final Exam, MC102 2nd Term 2016

MC102: Algorithms and Computer Programming

Institute of Computing – UNICAMP

[Instructor's name omitted] – Classes M and N

2nd Exam

Name:

Student ID:

Instructions: The length of the exam is 120 minutes. You may not consult any material. In the event of fraud, all of those involved will be given grade zero for the entire term. You may take this exam in pencil. Have a good test!

1. **(2.5)** Consider the Python code below. Assume that **v** is initialized with your six-digit student ID rather than with **123456**.

OBS: The correct printout of the first **print** command (content of **c**) is worth 1.0 points, whereas the correct printout of the contents of the list **v** is worth 1.5 points.

```
def main():
      v = [1, 2, 3, 4, 5, 6] # Replace with your student ID
      a = 2; b = 8; c = 1;

      for i in range(0, b):
            if(i % 2 == 0):
                  c = c * a

print("c␣=␣%d" %c)

fun1(v)
print("v = ", end="")
for i in range(6):
      print("%d," %v[i], end="")
print("")

def fun1(v):
      i = 0
      while (i < len(v)):
            v[i], v[i+1] = v[i+1], v[i]
            i = i + 2

main()
```

2. Write a function called **average**, which receives a list of numbers, and returns a tuple of length two, whose first position must contain the mean of the elements of the list and the second must contain the index of the element whose value is closest to the mean, in other words, the element of with index *i* is such that $|\text{list}[i] - \text{mean}|$ is the least possible.

The prototype of the function must be the following:

```
def average(my_list)
```

3. (**2.5**) Write a **recursive** function to find the largest element in a list of numbers that is located between the position 0 and a position **end**.

The prototype of the function must be the following.

```
def greatest(my_list, end);
```

4. (**2.5**) Consider the classes below:

```
class Student:
      def __init__(self):
            self.id = 0
            self.phone = 0
            self.name = ""

class Base:
      def __init__(self):
            self.students = []
```

Suppose we have a database represented as a class of the type **Base** where in the list **students** we store the data of students according to the class of type **Student**.

Write a method for the **Base** class which inserts/updates a student in the database **students**. The method receives as a parameter an object of type **Student** which already contains the student ID, Phone number, and name of a student. In case there already exists a student with the same ID as the one informed, then the student's name and phone number must be updated with the information passed as parameter. In case there is no student with the informed ID, then the program must include this student in the database. The prototype of the method should be:

```
def add(new_student)
```

Appendix C Detailed information on student answers

The following tables include the full record of misconceptions found and / or raised during the analyses of exams outlined in Section 3 and exemplified in Table 2. Each number in the columns marked *Unique Anonymized Identifier*, as the header suggests, corresponds to a unique student in all of the exams covered in this report.

Table 12: Students' answers in MC102/2/2015 Midterm

Section: MC102, 2nd term 2015, I1				
Exam: Midterm				
Unique Anonymized Identifier	**Question**			
	1	**2**	**3**	**4**
1	B3 D4	D2 A1	D5	O
2	O	D4	O	O
3	X1	D1	D2 D3	X3
4	X1	D2 D3	O	D2
5	X1	D2	D2	C
6	C	C	C	C
8	X1	O	X3	O
9	B1	D3	D6	O
11	X1	D2	C	O
12	X1	O	D5	N
13	B1	D1 D3 D4 D6	D3	O
15	O	O	D4	O
18	A3	D2	O	N
19	O	N	N	O
20	O	O	O	O
22	O	C	C	O
27	X1	C	D2	C
28	O	D2	N	O
29	O	D2	D2	O
30	O	D4	O	X2
31	O	D4	O	O
32	X1	D2 X2	O	O
36	B1 X1	D4	A5	A5
38	O	O	O	O
39	O	D5	D3	C
43	A3	D2	O	X3 A5
44	X1	D3	O	O
53	X1	D3	C	C
54	X1 B3	O	O	C
55	X1	O	C	C
56	X1	C	C	C
57	O	C	C	C
58	X1	D2 D6	D3	O
61	O	D3	C	C

Section: MC102, 2nd term 2015, I1				
Exam: Midterm				
Unique Anonymized Identifier	**Question**			
	1	**2**	**3**	**4**
62	O	D2	D2	O
63	X1	D3	D4	O
65	O	O	O	N
67	X1	C	O	C
69	O	D3	O	C
71	X1	D1 D2	D2	O
72	X1	D3 D2	O	D2
74	X1	D2	O	O
78	O	C	O	O
80	X1	C	C	O
81	B1	O	D3	C
83	O	D2	D2	X3
84	B1	N	D5	O
89	O	D1	O	D3
90	O	C	C	C
92	O	O	C	C
94	C	C	C	C
74	B3 D4	D2 A1	D5	O
78	O	D4	O	O
80	X1	D1	D2 D3	X3
81	X1	D2 D3	O	D2
83	X1	D2	D2	C
84	C	C	C	C
89	X1	O	X3	O
90	B1	D3	D6	O
92	X1	D2	C	O
94	X1	O	D5	N

Table 13: Student's Answers in Final MC102/2/2015

Section: MC102, 2nd term 2015, I1				
Exam: Final				
Unique Anonymized Identifier	**Question**			
	1	**2**	**3**	**4**
1	X4 X5	X2 X3	O	D5 X4
4	X2	C2 C3	D4	D4 D1
5	C	C2	D4	N
6	D3 X5 X7	C2	N	D3
8	X7	C1	O	D6
11	O	O	O	O
12	O	C	N	O
13	N	N	O	O
15	X4	O	O	X3

Section: MC102, 2nd term 2015, I1				
Exam: Final				
Unique Anonymized Identifier	**Question**			
	1	**2**	**3**	**4**
16	C	D3	X6	O
19	N	N	N	N
22	C	C	O	O
27	O	O	O	O
28	C	O	G2	O
29	X8	C2	D3	D1
30	C	C	O	C
31	O	C	O	O
36	D3 O	O	O	O
38	X5 O	O	O	X4
39	C	C	N	D3
43	O	C2	O	O
44	X6	C	N	O
53	C	D3	C	O
54	O	C	O	O
55	C	O	O	D3
56	O	C	O	D3
57	C	C	C	C
61	C	C	N	C
62	C	X2 C2	D4	X4
63	O	O	D3	D5
68	O	C2	O	D2
69	X5	C2	O	X4
71	O	C	O	D5
72	O	C	O	O
74	A4	O	O	C
78	C	C1 C3	O	C
80	X6	O	N	O
81	C	C	O	D3
89	X4 X5	O	D2	O
90	C	C	O	C
94	C	C	C	C

Table 14: Students' Answers in Retake MC102/2/2015

Section: MC102, 2nd term 2015, I1				
Exam: Retake				
Unique Anonymized Identifier	**Question**			
	1	**2**	**3**	**4**
1	O	O	C	O
4	C	O	D4	O
8	O	C	C	C
12	X1	C	N	O
15	O	C	O	C
16	O	C	C	C
29	O	C	C	X4
38	C	C	C	X4
43	X1	C	O	X2
44	O	C	N	C
63	X1	C	C	C
89	C	O	C	O

Table 15: Students' Answers in Midterm MC102/2/2016

Section: MC102, 2nd term 2016, I2				
Exam: Midterm				
Unique Anonymized Identifier	**Question**			
	1	**2**	**3**	**4**
7	C	C	O	C
14	O	C D4	N	C
17	O	X9	D6 X11	C
21	O	C	D3 X12	C
23	O	C	D3	C
24	O	X4	X4 X9	O
25	O	O	D6	O
26	O	D4	O	O
33	C	C	D4	O
34	O	O	D2 D4	O
35	C	D2	D1 X13	C
37	N	O	D4	O
40	O	O	O	O
41	C	C	O	C
42	O	D1	N	O
45	O	O	N	C
46	O	O	D1	C
47	C	D4 / D4	D6	O
48	O	X4	N	O
49	C	X10	D4	C
50	O	D1 D6	D3 X5	O
51	O	O	O	C

Section: MC102, 2nd term 2016, I2				
Exam: Midterm				
Unique Anonymized Identifier	**Question**			
	1	**2**	**3**	**4**
52	C	O	C	C
59	O	O	D4	C
60	O	D2 D4	D2 X13	C
64	O	O	D4	C
66	O	O	C	C
70	O	O	D4	C
73	O	D2 D4	O	O
75	C	C	D3	C
76	C	C	D6	C
77	O	O	O	O
79	O	O	N	C
82	O	C	C	O
85	O	D5	O	C
86	O	O	N	O
88	O	X9 D1 D2	D4	O
91	O	O	O	C
93	O	O	O	O
95	C	D4	C	C
96	N	O	O	O
97	O	X6	O	O
98	O	C	C	C
99	C	C	C	C
100	O	H3	D1 D4	O
101	C	D6 D5	N	C
102	C	D5	N	O
103	C	C	D4	O
104	O	C	C	C
105	C	C	D4	C
106	C	C	D4	C
107	C	G1	O	C
108	N	O	O	C

Table 16: Students' Answers in Final MC102/2/2016

Section: MC102, 2nd term 2016, I2				
Exam: Final				
Unique Anonymized Identifier	**Question**			
	1	**2**	**3**	**4**
10	C	O	C	C
17	C	C	C	N
21	C	C	C	C
23	O	O	A5	X14
24	O	O	O	X15

Section: MC102, 2nd term 2016, I2				
Exam: Final				
Unique Anonymized Identifier	**Question**			
	1	**2**	**3**	**4**
25	C	O	N	C
33	C	C	D4 C3	C
34	C	N	O	X16
35	C	C	O	C
37	O	O	C1	D2
40	C	O	O	X7
41	C	C	O	O
46	O	C	C	C
47	C	O	O	D2
48	C	C	C	C
49	C	C	C	C
50	O	O	C2	X15
51	C	O	C	C
52	O	O	C	N
59	C	O	O	C
60	C	C	C	C
66	O	D4	O	X15
70	O	O	O	D2
73	C	N	O	D2
75	C	C	O	X15
76	C	O	O	D2
77	O	O	O	O
79	C	O	C	D2
82	C	C	C	C
85	C	O	C	C
87	C	N	N	O
91	O	O	C1	D2
93	C	D5 A1	A1 C2	D2
95	C	O	C	X17
96	C	O	O	D2
97	C	C	C	D2
98	C	C	O	X15
99	C	O	C	O
100	C	O	C	D2
101	C	C	C	O
102	C	C	O	D2
103	C	O	C	D2
104	C	O	O	D2
105	C	N	C	O
106	C	C	C	D2
107	C	O	C	X17 D2
108	O	C	O	D2

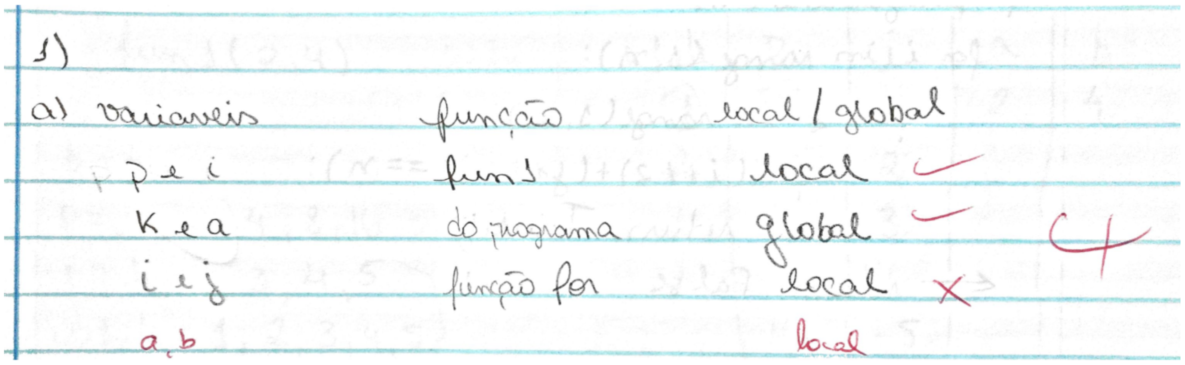
Appendix D Collected samples of student answers

This appendix contains scans of student answers from the analyzed exams that we considered to illustrate best the new hypothetical misconceptions we've raised as outlined in Section 3. Table 17 maps each figure to the misconception it illustrates, and to a comment on the student's response.

Table 17: Index of the excerpts shown in Figures 15-26, mapping them to hypothetical misconceptions found in this work.

Misconception	**Figure**	**Comments**
PB.5	15	Student deems variables *i* and *j* as local to a "function" *for*.
PB.6	16	Variable *a* receives return value from function *perf* before the arguments are initialized.
PD.7	17	variable *i* initialized for list elements rather than index
PD.8	18	variable *j* initialized as index but used as list element
PH.1	19	Attempt to use *append* method on *tuple* instance.
PH.1	20	Attempt to use *split* method on *int*.
PH.2	21	*list* object initialized, but not attributed to variable
PH.3	22	*maior* function invoked preceded by *def* keyword.
PH.4	23	*join* method invoked with no instance of *str* specified.
PH.5	24	attempt to attribute value to *append* method
PH.6	25	No use of *self* keyword to reference instance attributes
PH.7	26	Instance attribute *alunos* is a *list* object containing

Misconception	Figure	Comments
		instance of class *Aluno*. Student attempts to access attributes of *Aluno* object through square-bracket indexing.

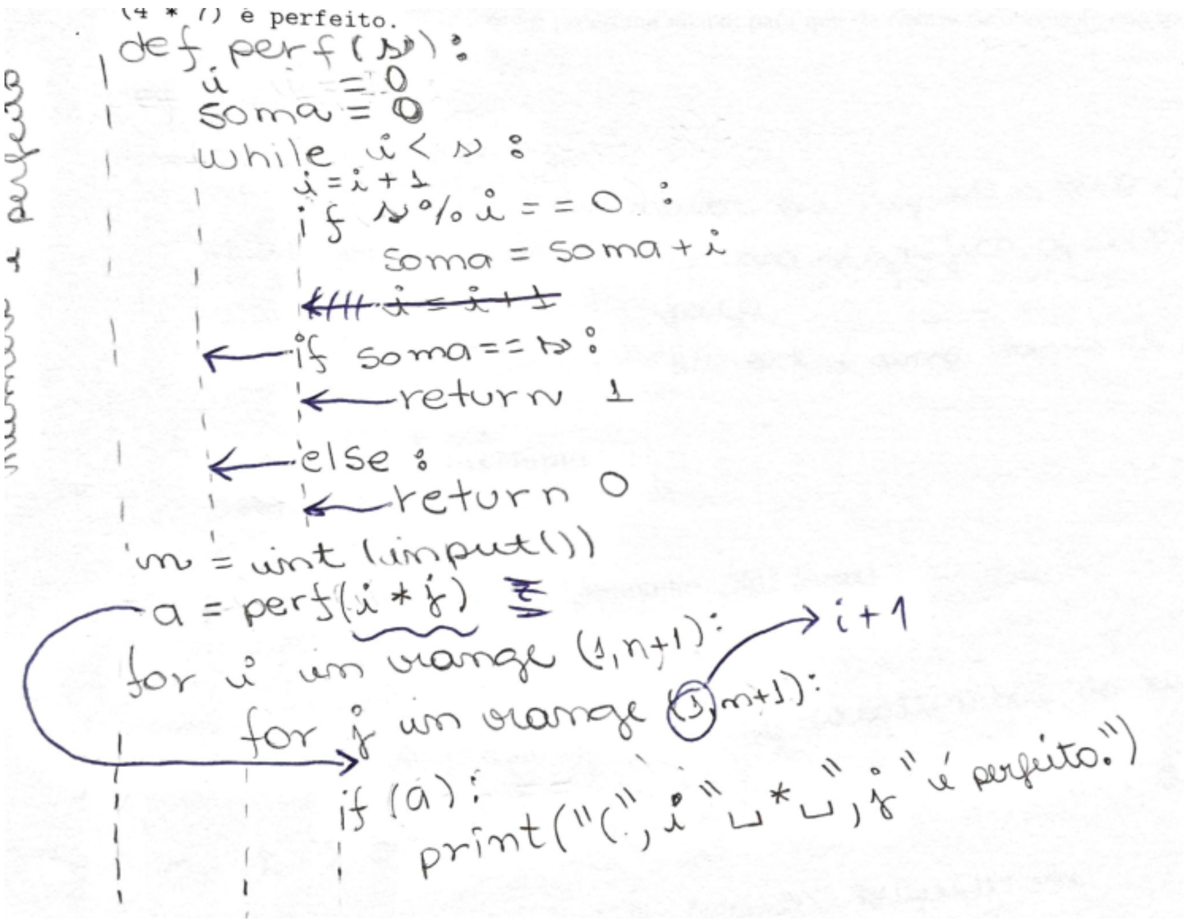


1)

a) variaveis	função	local / global	
p e i	fun1	local	✓
k e a	do programa	global	✓
i e j	função for	local	✗
a, b		local	

C+

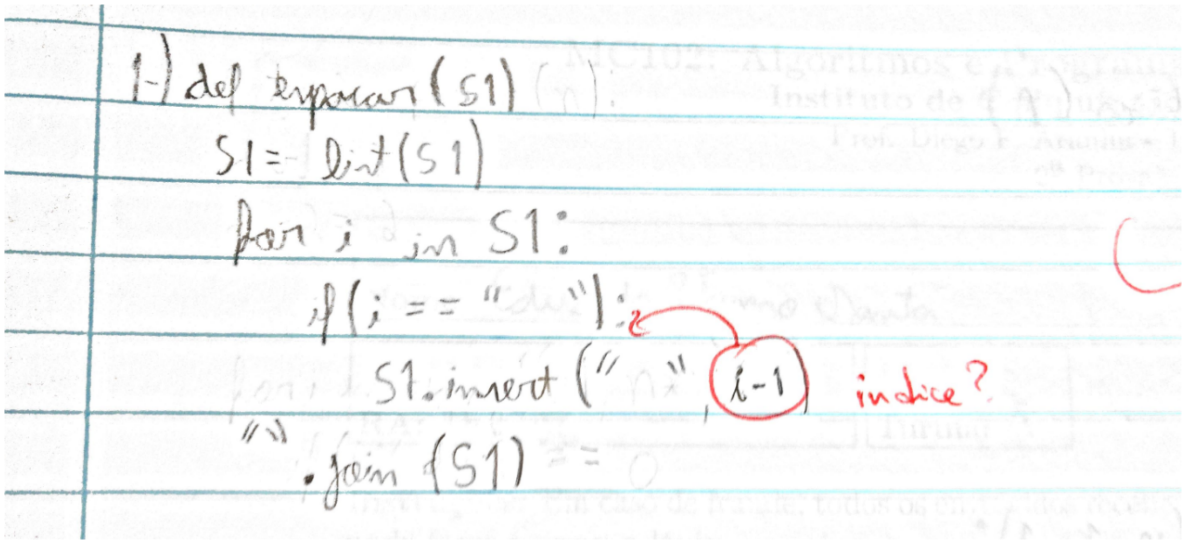
Figure 15: Misconception PB.5 (Retake/2/2015)



```
(4 * 7) é perfeito.
def perf(s):
    i = 0
    soma = 0
    while i< s:
        i=i+1
        if s%i == 0:
            soma = soma+i
        ~~i=i+1~~
    if soma==s:
        return 1
    else:
        return 0
n = int(input())
a = perf(i*j)
for i in range(1,n+1):        i+1
    for j in range(1,n+1):
        if (a):
            print("(", i " * ", j " é perfeito.")
```

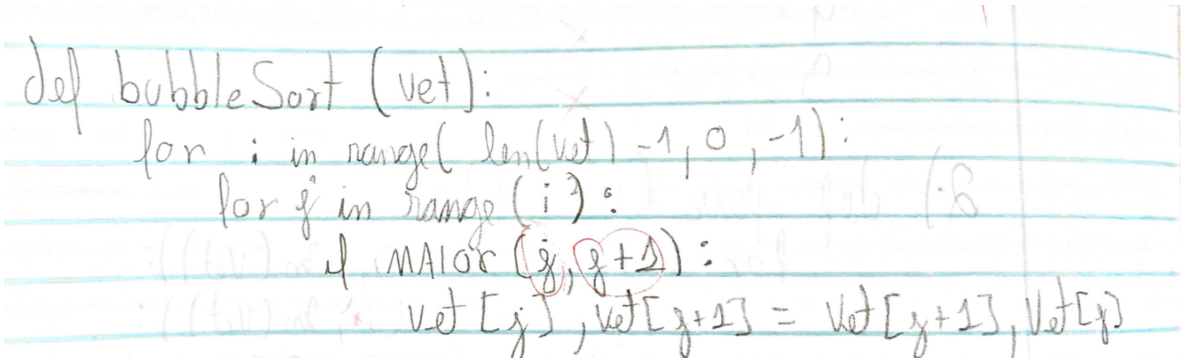
número é perfeito

Figure 16: Misconception PB.6 (Midterm/2/2016):



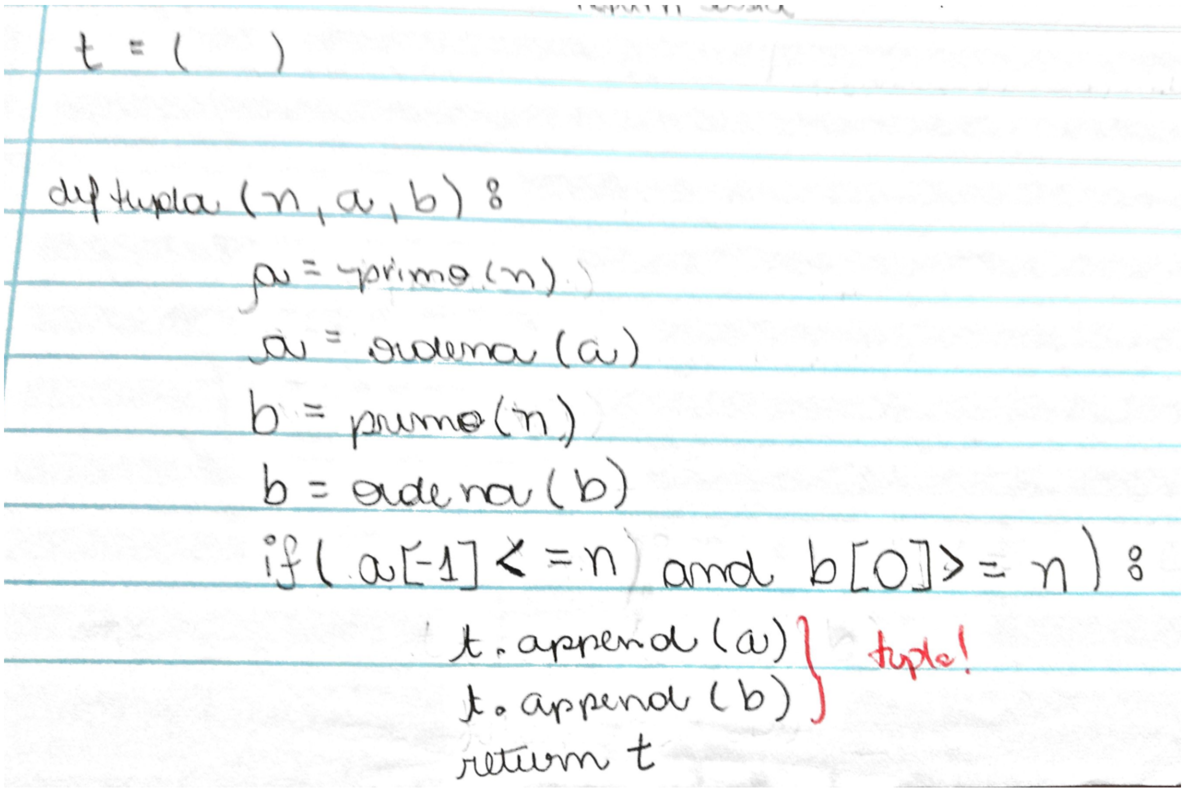
```
1-) def espacar(S1)
    S1= list(S1)
    for i in S1:
        if(i == "dv"):
            S1.insert("\n", i-1)   indice?
    "".join(S1)
```

Figure 17: Misconception PD.7 (Final/2/2015):



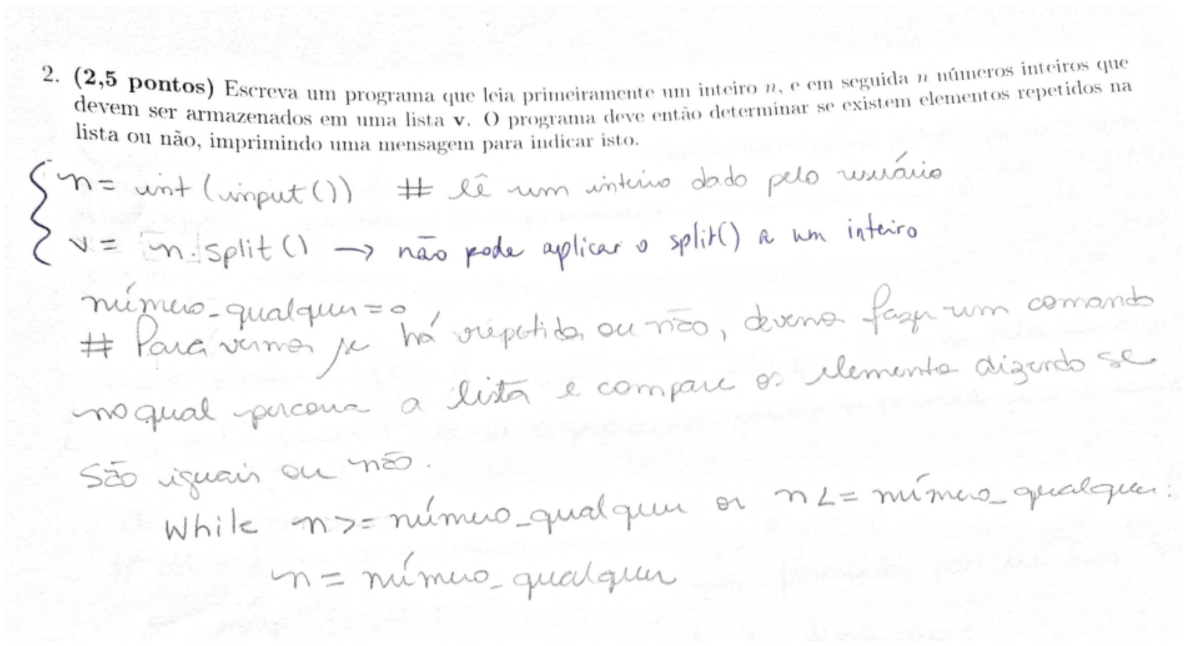
```
def bubbleSort (vet):
    for i in range( len(vet) -1, 0, -1):
        for j in range(i):
            if MAIOR(j, j+1):
                vet[j], vet[j+1] = vet[j+1], vet[j]
```

Figure 18: Misconception PD.8 (Midterm/2/2015):



```
t = (   )

def tupla(n, a, b):
        a = primo(n)
        a = ordena(a)
        b = primo(n)
        b = ordena(b)
        if( a[-1] <= n) and b[0]>= n):
            t.append(a)
            t.append(b)
            return t
```

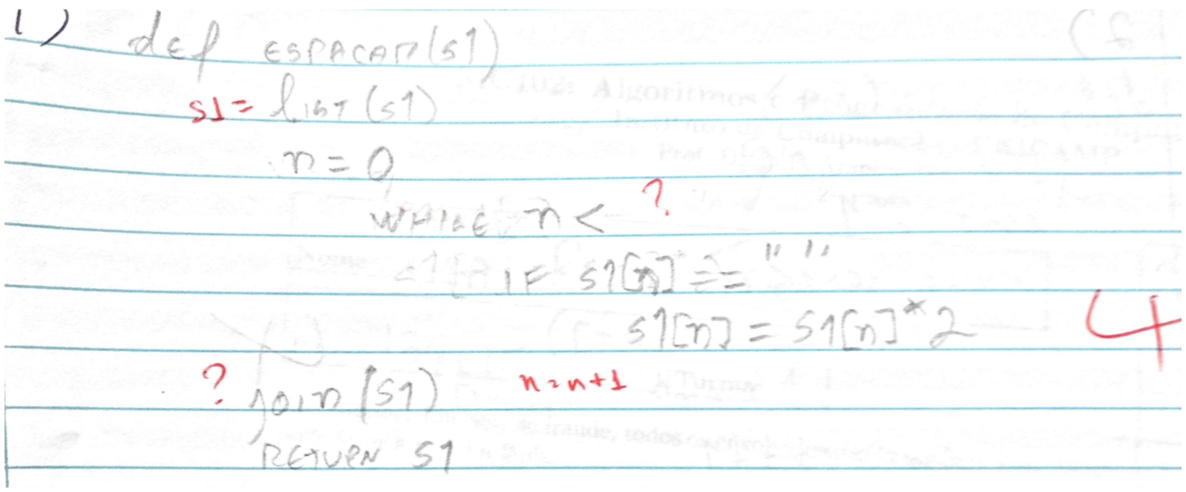
Figure 19: Misconception PH.1 (Final/2/2015):



2. **(2,5 pontos)** Escreva um programa que leia primeiramente um inteiro n, e em seguida n números inteiros que devem ser armazenados em uma lista **v**. O programa deve então determinar se existem elementos repetidos na lista ou não, imprimindo uma mensagem para indicar isto.

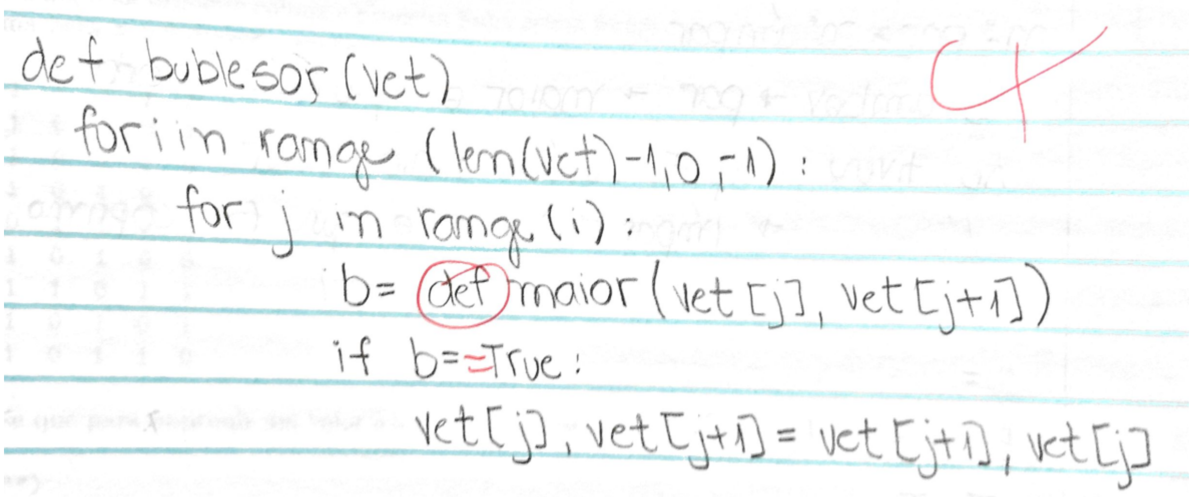
```
n = int(input())   # lê um inteiro dado pelo usuário
v = n.split() → não pode aplicar o split() a um inteiro

número_qualquer = 0
# Para vermos se há repetido, ou não, devemos fazer um comando
no qual percorra a lista e compare os elementos dizendo se
são iguais ou não.
    While n >= número_qualquer or n <= número_qualquer:
        n = número_qualquer
```

Figure 20: Misconception PH.1 (Midterm/2/2016):



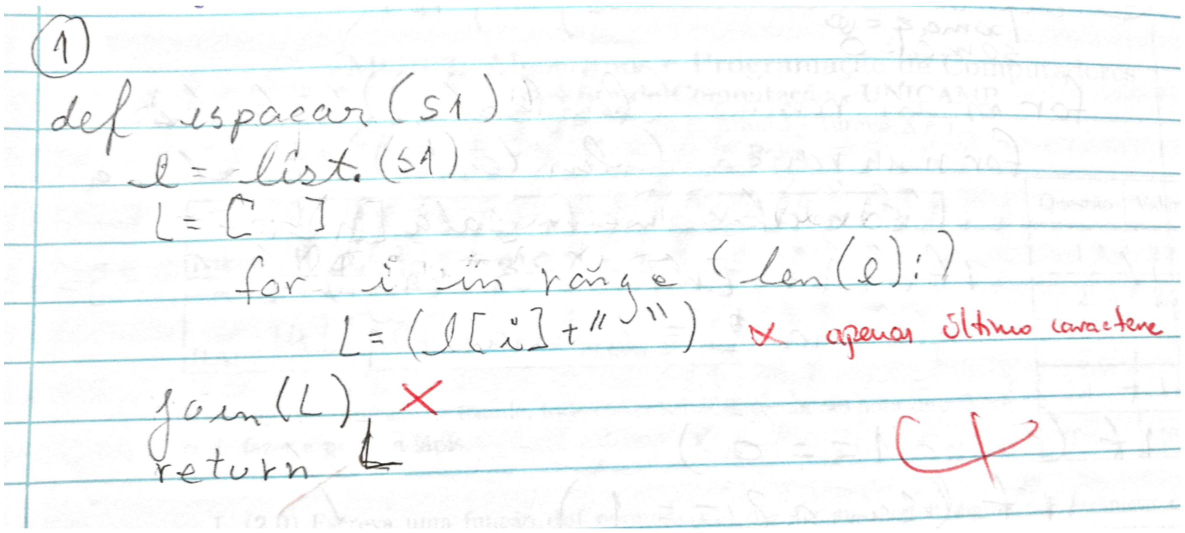
```
1)  def espacar(s1)
        s1 = list(s1)
        n = 0
            while n <  ?
          s1[n] if s1[n] == " "
                s1[n] = s1[n]*2
    ?   join(s1)      n=n+1
        return s1
                                              4
```

Figure 21: Misconception PH.2 (Final/2/2015):



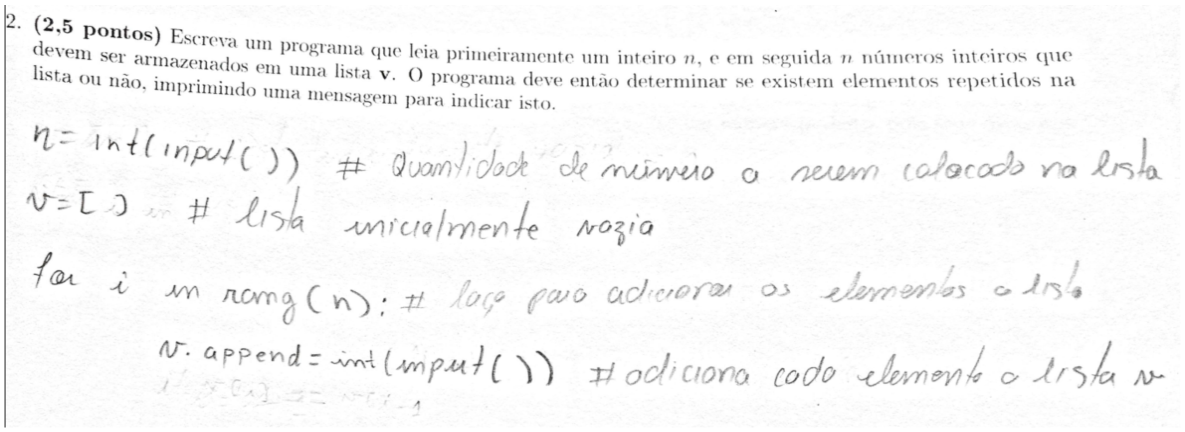
```
def bublesor(vet)
  for i in range (len(vet)-1,0,-1):
     for j in range(i):
          b= def maior(vet[j], vet[j+1])
          if b==True:
             vet[j], vet[j+1] = vet[j+1], vet[j]
```

Figure 22: Misconception PH.3 (Midterm/2/2015):



```
(1)
def espacar(s1)
   l = list.(s1)
   L = [ ]
      for i in range(len(l):)
         L = (l[i]+" ")   × apenas último caractere
   join(L) ×
   return L
```

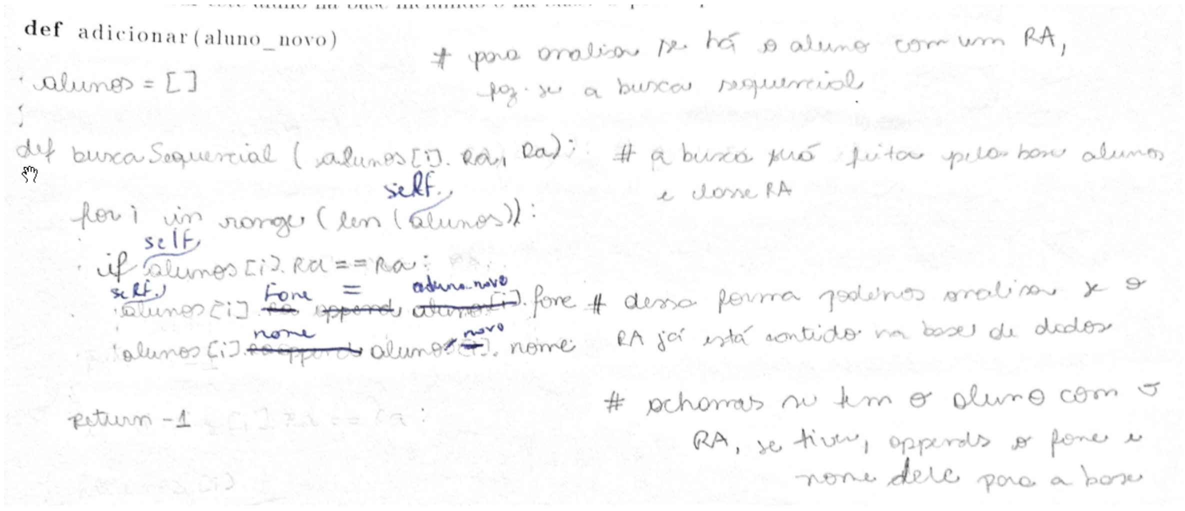
Figure 23: Misconception PH.4 (Final/2/2015):



2. **(2,5 pontos)** Escreva um programa que leia primeiramente um inteiro n, e em seguida n números inteiros que devem ser armazenados em uma lista **v**. O programa deve então determinar se existem elementos repetidos na lista ou não, imprimindo uma mensagem para indicar isto.

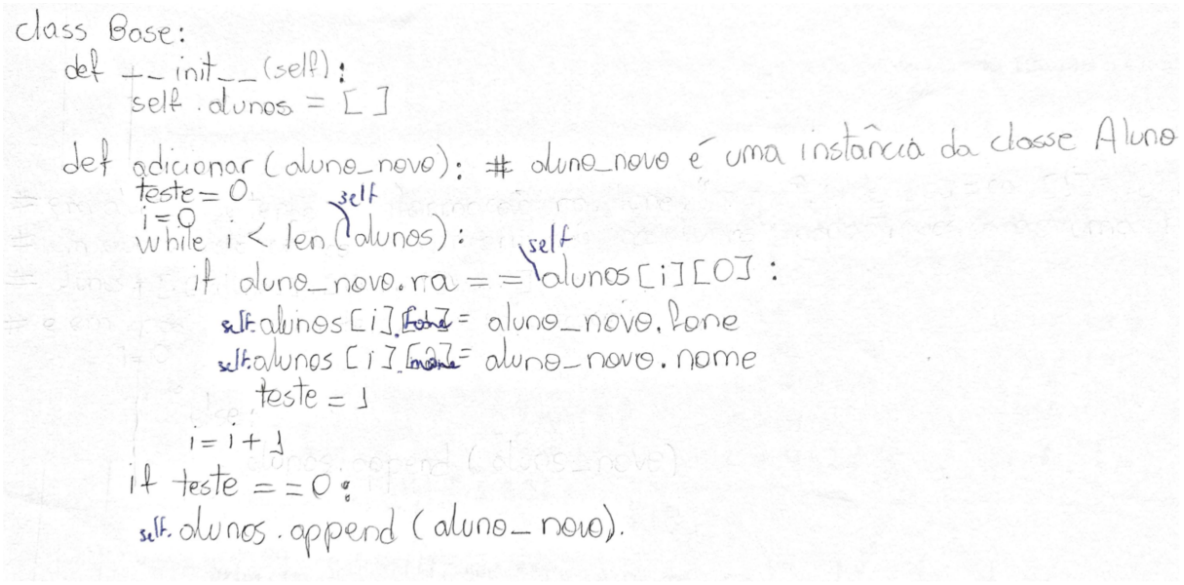
```
n= int(input()) # Quantidade de número a serem colocados na lista
v=[] # lista inicialmente vazia
for i in range(n): # laço para adicionar os elementos a lista
    v.append= int(input()) #adiciona cada elemento a lista v
```

Figure 24: Misconception PH.5 (Midterm/2/2016):



```
def adicionar(aluno_novo)
                                   # para analisar se há o aluno com um RA,
alunos = []                           faz-se a busca sequencial

def buscaSequencial ( alunos[i]. Ra, Ra): # a busca será feita pela base alunos
                                             e desse RA
                  self.
   for i in range ( len (alunos)):
      self.
    if alunos[i]. Ra == Ra:
   self.            fone  =    aluno_novo
      alunos[i]. fo append alunos[i]. fone # dessa forma podemos analisar se o
                  nome               novo
      alunos[i]. reappend alunos[i]. nome   RA já está contido na base de dados

                                      # achamos se tem o aluno com o
   return -1                             RA, se tiver, appends o fone e
                                           nome dele para a base
```

Figure 25: Misconception PH.6 (Final/2/2016):



```
class Base:
    def __init__(self):
        self.alunos = [ ]

    def adicionar(aluno_novo): # aluno_novo é uma instância da classe Aluno
        teste = 0
        i = 0
        while i < len(self.alunos):
            if aluno_novo.ra == self.alunos[i][0]:
                self.alunos[i]~~[1]~~.fone = aluno_novo.fone
                self.alunos[i]~~[2]~~.nome = aluno_novo.nome
                teste = 1
            i = i + 1
        if teste == 0:
            self.alunos.append(aluno_novo).
```

Figure 26: Misconception PH.7 (Final/2/2016):