python-by-contract-corpus Documentation

Release 2021.7.10rc1

Lauren De bruyn, Marko Ristin, Phillip Schanely

Jun 13, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

We present here a corpus of Python programs annotated with contracts.

The corpus includes:

- Solutions to the exercises of the Advent of Code 2020,
- Solutions to the exercises used during the lecture "Introduction to Programming" at ETH Zurich (Switzerland) in Fall 2019, and
- Incorrect code with minimal difference to the final solutions obtained from recorded failures (see *Recorded Failures* and *Incorrect Programs*).

The design-by-contract is still not widely practiced in Python due to different factors such as unfamiliarity of the community with the concept and with the available tools. Therefore we could not find a sufficiently large and representative code base that would give us a good testbed for the automatic tools.

Hence we employ the solutions to the aforementioned exercises as a **benchmark** to compare and evaluate different approaches to automatic testing of Python code.

We expect this data set to help us **discover the blind spots**. The tools (usually) generate the input for the functions based on the type annotations and pre-conditions. So the corpus helps us answer the questions such as:

- Which families of pre-conditions are supported?
- What are the limits of a testing tool and which pre-conditions are not supported?
- Which kinds of inputs can be generated in a computationally efficient way?

We hope that this code base benefits not only the community of **tool developers** but also the **tool users** by exposing the involved **trade-offs**. In particular, the users should learn based on practical examples how the tools differ, understand their limits and distinguish in which cases one tool is better than the other.

CHAPTER

TWO

CORRECT PROGRAMS

As an important part of our corpus, we provide programs annotated with contracts with no obvious bugs.

2.1 Common

This is a module containing common functions shared among the different solutions. While we tried to make solutions as stand-alone as possible, we could not help but encapsulate a couple of patterns to help readability.

Provide common functionality shared among all the solutions.

Classes:

Lines(lines)	Represent a sequence of text lines.

Functions:

pairwise(iterable)	Iterate over `(s0, s1, s2, ...)` as `((s0, s1), (s1, s2), ...)`.

class Lines(*lines: Sequence[str]*)

Represent a sequence of text lines.

Methods:

__new__(cls, lines)	Ensure the properties on the `lines`.
__add__(other)	Concatenate two list of lines.
__getitem__()	Get the line(s) at the given index.
__len__()	Return the number of the lines.
__iter__()	Iterate over the lines.

static __new__(*cls*, *lines: Sequence[str]*) → *Lines*

Ensure the properties on the `lines`.

Please make sure that you transfer the "ownership" immediately to Lines and don't modify the original list of strings any more:

```
##
# OK
##
```

(continues on next page)

(continued from previous page)

```
lines = Lines(some_text.splitlines())

##
# Not OK
##

some_lines = some_text.splitlines()
lines = Lines(some_lines)
# ... do something assuming ``lines`` is immutable ...

some_lines[0] = "This will break \n your logic"
# ERROR! lines[0] now contains a new-line which is unexpected!
```

Requires

- `all('n' not in line and 'r' not in line for line in lines)`

__add__(*other:* Lines) → *Lines*

Concatenate two list of lines.

__getitem__(*index: int*) → str

__getitem__(*index: slice*) → *Lines*

Get the line(s) at the given index.

__len__() → int

Return the number of the lines.

__iter__() → Iterator[str]

Iterate over the lines.

pairwise(*iterable: Iterable[T]*) → Iterable[Tuple[T, T]]

Iterate over `(s0, s1, s2, ...)` as `((s0, s1), (s1, s2), ...)`.

```
>>> list(pairwise([]))
[]
```

```
>>> list(pairwise([1]))
[]
```

```
>>> list(pairwise([1, 2]))
[(1, 2)]
```

```
>>> list(pairwise([1, 2, 3]))
[(1, 2), (2, 3)]
```

2.2 Solutions to Advent of Code 2020

We auto-document here our solutions to the Advent of Code 2020 exercises.

The source code of our solutions is available at: https://github.com/mristin/python-by-contract-corpus/tree/main/python_by_contract_corpus/correct/aoc2020

2.2.1 Day 1: Report Repair

Functions:

find_pair_with_sum(items, total)	Find the two entries that sum to `total`.

find_pair_with_sum(*items: List[int]*, *total: int*) → Optional[Tuple[int, int]]

Find the two entries that sum to `total`.

Ensures

- ```
  result is None
  or result[0] != result[1]
  or items.count(result[0]) > 1
  ```

  (A duplicated result was produced from different input items)

- `result is not None all(r in items for r in result)`

  (Returned values appear in the input)

- `result is not None result[0] + result[1] == total`

  (Returned items sum to the right total)

2.2.2 Day 2: Password Philosophy

Functions:

verify_line(line)	Verify an entry of the password database.
verify(min_count, max_count, character, password)	Verify the `password` under the given constraints.

verify_line(*line: str*) → Optional[bool]

Verify an entry of the password database.

Requires

- `ENTRY_RE.match(line)`

verify(*min_count: int*, *max_count: int*, *character: str*, *password: str*) → bool

Verify the `password` under the given constraints.

Requires

- `len(character) == 1`
- `min_count <= max_count`
- `max_count > 0`
  ```

- `min_count > 0`

Ensures

- `len(password) == 0 not result`

2.2.3 Day 3: Toboggan Trajectory

Classes:

InputLine(line)	Represent a line of the input map.

Functions:

parse_input(lines)	Parse the input map given as `lines`.
count_trees(width, height, input_string)	Count the trees in the `input_string`.

class InputLine(*line: str*)

Represent a line of the input map.

parse_input(*lines: Sequence[InputLine]*) → Tuple[int, int, str]

Parse the input map given as `lines`.

Requires

- `len(lines) > 0`
- ```
  all(
      len(line) == len(lines[0])
      for line in lines
  )
  ```

Ensures

- `result[0] * result[1] == len(result[2])`

count_trees(*width: int*, *height: int*, *input_string: str*) → int

Count the trees in the `input_string`.

Requires

- `width * height == len(input_string)`
- `re.match(r"^[.#]*", input_string)`
- `width > 0 and height > 0`

Ensures

- `result <= height / STEP_SIZE_VERTICAL`
  ```

2.2.4 Day 4: Passport Processing

Functions:

blank_line_split(text)	Split the text on a blank line.
parse_passport_entries(text)	Parse the passport entries separated by a blank-line in `text`.
is_valid(entry)	Verify whether the passport entry is valid.
count_valid(batch)	Count the number of valid passports in the `batch`.

blank_line_split(*text: str*) → List[str]

Split the text on a blank line.

```
>>> blank_line_split('X\nY\n\nZ')
['X\nY', 'Z']
```

Ensures

- `len(result) == text.count("nn") + 1`

parse_passport_entries(*text: str*) → List[Tuple[str, str]]

Parse the passport entries separated by a blank-line in `text`.

Requires

- `PASSPORT_RE.fullmatch(text)`

is_valid(*entry: List[Tuple[str, str]]*) → bool

Verify whether the passport entry is valid.

Ensures

- `result == all(k in dict(entry) for k in _REQUIRED_KEYS)`

count_valid(*batch: str*) → int

Count the number of valid passports in the `batch`.

Requires

- `all(PASSPORT_RE.match(line) for line in blank_line_split(batch))`

Ensures

- `result >= 0`

2.2.5 Day 5: Binary Boarding

Functions:

apply(first, last, directive)	Apply the `directive` given the range as `[first, last]` (inclusive).
determine_row(identifier)	Compute the row of the seat identified by the `identifier`.
determine_column(identifier)	Compute the column of the seat identified by the `identifier`.
determine_row_and_column(identifier)	Compute the `(row, column)` of the seat identified by the `identifier`.
determine_id(row, column)	Compute the identifier of the seat given its `row` and `column`.

apply(*first: int*, *last: int*, *directive: str*) → Tuple[int, int]

Apply the `directive` given the range as `[first, last]` (inclusive).

Returns

new first, new last

Requires

- `re.match(r"^[FBLR]Z", directive)`
- `first >= 0`
- `last >= 1`
- `first % 2 == 0`
- `last % 2 == 1`
- `first < last`
- `(last - first + 1) % 2 == 0`

  (Range always divisible by 2)

Ensures

- `result[0] != result[1] result[1] % 2 == 1`

  (next last always not divisible by 2 unless the last step)

- `result[0] != result[1] result[0] % 2 == 0`

  (next first always divisible by 2 unless the last step)

- `first <= result[0] <= result[1] <= last`
- ```
  not (directive in "BR")
  or (first < result[0] and result[1] == last)
  ```

  (Only first increases on B/R.)

- ```
  not (directive in "FL")
  or (result[0] == first and result[1] < last)
  ```

  (Only last decreases on F/L.)

determine_row(*identifier: str*) → int

Compute the row of the seat identified by the `identifier`.

Requires

- `re.match(r"^[FB]{7}Z", identifier)`

Ensures

- `0 <= result <= 127`

determine_column(*identifier: str*) → int

Compute the column of the seat identified by the `identifier`.

Requires

- `re.match(r"^[LR]{3}Z", identifier)`

Ensures

- `0 <= result <= 127`

determine_row_and_column(*identifier: str*) → Tuple[int, int]

Compute the `(row, column)` of the seat identified by the `identifier`.

Requires

- `re.match(r"^[FB]{7}[LR]{3}Z", identifier)`

Ensures

- `0 <= result[1] <= 8`
- `0 <= result[0] <= 127`

determine_id(*row: int*, *column: int*) → int

Compute the identifier of the seat given its `row` and `column`.

Requires

- `0 <= column <= 8`
- `0 <= row <= 127`

Ensures

- `result <= 127 * 8 + 8`

  (The highest seat ID)

2.2.6 Day 6: Custom Customs

Functions:

solve(input_string)	Count the number of yes answers in the group given as `input_string`.

solve(*input_string: str*) → int

Count the number of yes answers in the group given as `input_string`.

Requires

- ```
  all(
      len(set(list(line))) == len(line)
      for line in input_string.split("\n")
  )
  ```
  ```

- ```
  all(
      re.match(r"^[a-z]*$", line)
      for line in input_string.split("\n")
  )
  ```

Ensures

- `result >= 0`

2.2.7 Day 7: Handy Haversacks

Functions:

parse_bagexpr(text)	Parse the expression representing a bag.
parse_rule(text)	Parse the rule from the text given as a single line.
parse_rules(lines)	Parse the rules from the given `lines`.
directly_contains(container, contained, rules)	Check whether the `contained` is contained in the `container`.
containers(kind, rules)	Compute the set of known containers.
count_containers(lines)	Parse the rules given as `lines` and count the allowed containers.
main()	Execute the main routine.

parse_bagexpr(*text: str*) → Tuple[int, str]

Parse the expression representing a bag.

Returns

number of units, kind

Requires

- `re.fullmatch(r"d+ .* bags?", text)`

parse_rule(*text: str*) → Tuple[str, Dict[str, int]]

Parse the rule from the text given as a single line.

Returns

kind, contents

Requires

- `"n" not in text`

parse_rules(*lines:* Lines) → Dict[str, Dict[str, int]]

Parse the rules from the given `lines`.

directly_contains(*container: str*, *contained: str*, *rules: Dict[str, Dict[str, int]]*) → bool

Check whether the `contained` is contained in the `container`.

containers(*kind: str*, *rules: Dict[str, Dict[str, int]]*) → Set[str]

Compute the set of known containers.

Ensures
  ```

- ```
  all(
      not directly_contains(non_container, container, rules)
      for non_container in (rules.keys() - result)
      for container in result
  )
  ```

  (Nothing else contains anything in the result)

- `kind in result`

  (given kind is in the result)

count_containers(*lines:* Lines) → int

Parse the rules given as `lines` and count the allowed containers.

main() → None

Execute the main routine.

2.2.8 Day 8: Handheld Halting

Classes:

Operation(value)	Represent an operation corresponding to an instruction.
Instruction(operation, argument)	Represent an instruction of the boot code.

Functions:

parse_line(line)	Parse a `line` of the boot code into an instruction.
parse(lines)	Parse the boot code given as `lines`.
execute_instructions(instructions)	Execute the boot code given as `instructions`.

class Operation(*value*)

Represent an operation corresponding to an instruction.

Attributes:

NOP
ACC
JMP

NOP = 'nop'

ACC = 'acc'

JMP = 'jmp'

class Instruction(*operation:* Operation, *argument: int*)

Represent an instruction of the boot code.

Attributes:
  ```

operation	the corresponding operation
argument	the argument to the operation

operation: *Operation*

the corresponding operation

argument: **int**

the argument to the operation

parse_line(*line: str*) → *Instruction*

Parse a `line` of the boot code into an instruction.

Requires

- `INSTRUCTION_RE.match(line)`

parse(*lines:* Lines) → List[*Instruction*]

Parse the boot code given as `lines`.

Requires

- `all(INSTRUCTION_RE.match(line) for line in lines)`

Ensures

- `len(lines) == len(result)`

execute_instructions(*instructions: List[*Instruction*]*) → Optional[int]

Execute the boot code given as `instructions`.

Returns

The value in the accumulator just before an instruction is run for the second time

Requires

- ```
  all(
      0 <= i + instruction.argument < len(instructions)
      for i, instruction in enumerate(instructions)
      if instruction.operation == Operation.JMP
  )
  ```

2.2.9 Day 9: Encoding Error

Functions:

solve(puzzle_input, preamble_length)	Parse the data of the XMAS protocol, `puzzle_input`, and find a weakness.

solve(*puzzle_input: List[int]*, *preamble_length: int*) → Optional[Tuple[int, int]]

Parse the data of the XMAS protocol, `puzzle_input`, and find a weakness.

Returns

offset of the number, first number *after* the preamble which uncovers the weakness

Requires

- `all(number >= 0 for number in puzzle_input)`
  ```

- `len(puzzle_input) > preamble_length`

Ensures

- ```
  not result
  or result[1]
  not in [
      sum(t)
      for t in combinations(puzzle_input[result[0] - preamble_length :␣
  ↪result[0]], 2)
  ]
  ```

- `result result[1] in puzzle_input`

2.2.10 Day 10: Adapter Array

Classes:

HistogramOfDeltas(mapping)	Represent a histogram of differences between adapters in jolts.

Functions:

histogram_differences(adapters)	Compute the histogram of jolt differences in `adapters`.
compute_result(histo)	Analyze the histogram of jolt differences.
parse(lines)	Parse the specification of the adapters given as `lines`.

class HistogramOfDeltas(*mapping: Mapping[int, int]*)

Represent a histogram of differences between adapters in jolts.

Methods:

__new__(cls, mapping)	Enforce histogram constraints.
__getitem__(key)	Retrieve the count corresponding to the histogram bin given with `key`.
__iter__()	Iterate over histogram bins.
__len__()	Retrieve the number of the histogram bins.

static __new__(*cls*, *mapping: Mapping[int, int]*) → *HistogramOfDeltas*

Enforce histogram constraints.

Requires

- `all(count >= 0 for count in mapping.values())`
- `all(delta > 0 for delta in mapping.keys())`

__getitem__(*key: int*) → int

Retrieve the count corresponding to the histogram bin given with `key`.

__iter__() → Iterator[int]

Iterate over histogram bins.
  ```

`__len__`() → int

Retrieve the number of the histogram bins.

`histogram_differences`(*adapters: List[int]*) → *HistogramOfDeltas*

Compute the histogram of jolt differences in `adapters`.

Requires

- `len(adapters) > 0`
- `0 not in adapters`

  (The charging input not in adapters)
- `all(adapter >= 0 for adapter in adapters)`
- `len(set(adapters)) == len(adapters)`

Ensures

- `len(adapters) == 0 sum(result.values()) == 0`

  (Empty histogram on empty input)
- `sum(result.values()) == len(adapters) + 1`

`compute_result`(*histo:* HistogramOfDeltas) → int

Analyze the histogram of jolt differences.

Returns

the product of the respective counts of 1s and 3s differences

Ensures

- `result >= 0`

`parse`(*lines:* Lines) → List[int]

Parse the specification of the adapters given as `lines`.

Returns

List of corresponding number of adapter jolts

Requires

- `all(re.match(r"^(0|[1-9][0-9]*)Z", line) for line in lines)`

Ensures

- `len(lines) == len(result)`

2.2.11 Day 11: Seating System

Functions:

list_neighbourhood(i, j, height, width)	List all the neighbours of the given seat at position `i, j`.
apply(layout)	Compute a single iteration.
apply_until_stable(layout)	Run the simulation until the layout does not change anymore.
parse_layout(lines)	Parse the initial layout given as `lines`.
repr_layout(layout)	Represent the layout as string for debugging.
count_occupied(layout)	Count the number of occupied seats in the `layout`.

Classes:

Layout(table)	Represent a seat layout.

list_neighbourhood(*i: int*, *j: int*, *height: int*, *width: int*) → List[Tuple[int, int]]

List all the neighbours of the given seat at position `i, j`.

The `height` and `width` define the limits of the layout.

Requires

- `0 <= j <= width`
- `0 <= i <= height`

Ensures

- `(i, j) not in result`
- `len(result) <= 8`
- ```
  all(
      0 <= i <= height and 0 <= j <= width for i, j in result
  )
  ```

class Layout(*table: List[List[str]]*)

Represent a seat layout.

Methods:

__init__(table)	Initialize with the given values.

Attributes:

table	matrix of the layout
height	height of the layout
width	width of the layout

__init__(*table: List[List[str]]*) → None

Initialize with the given values.

Requires

- ```
  all(
      re.fullmatch(r"[L#.]", cell)
      for row in table
      for cell in row
  )
  ```
- ```
  len(table) > 0
  and len(table[0]) > 0
  and all(
      len(row) == len(table[0])
      for row in table
  )
  ```
  ```

Ensures

- `len(self.table) > 0 and self.width == len(self.table[0])`
- `self.height == len(self.table)`

table: Final[List[List[str]]]

matrix of the layout

height: Final[int]

height of the layout

width: Final[int]

width of the layout

apply(*layout:* Layout) → Tuple[*Layout*, int]

Compute a single iteration.

Returns

(new layout, number of changes)

Ensures

- ```
  all(
      (cell == "." and result_cell == ".")
      or (cell != "." and result_cell in ["L", "#"])
      for row, result_row in zip(layout.table, result[0].table)
      for cell, result_cell in zip(row, result_row)
  )
  ```

  (Valid change)

- `layout.width == result[0].width`
- `layout.height == result[0].height`

apply_until_stable(*layout:* Layout) → *Layout*

Run the simulation until the layout does not change anymore.

parse_layout(*lines: List[str]*) → *Layout*

Parse the initial layout given as `lines`.

Requires

- ```
  not (len(lines) > 0)
  or all(len(line) == len(lines[0]) for line in lines)
  ```

  (Lines are a table)

- `all(re.match(r"^[.L#]+Z", line) for line in lines)`

Ensures

- ```
  not len(lines) == 0
  or all(len(line) == len(row) for line, row in zip(lines, result))
  ```

- `len(lines) == result.height`

repr_layout(*layout:* Layout) → str

Represent the layout as string for debugging.
  ```

count_occupied(*layout:* Layout) → int

Count the number of occupied seats in the `layout`.

2.2.12 Day 12: Rain Risk

Input: list of actions Output: position, Manhattan distance

Classes:

Orientation(value)	Represent the facing orientation of the ship.
ShipPosition(horizontal, vertical, orientation)	Represent the current ferry position.

Functions:

parse_input(puzzle_input)	Split the puzzle input along the newlines.
update_position(current_position, move)	Execute a single `move` on `current_position` and return a new position.
solve(puzzle_input)	Execute the instructions and return the final ship's position.

class Orientation(*value*)

Represent the facing orientation of the ship.

Attributes:

EAST
SOUTH
WEST
NORTH

EAST = 0

SOUTH = 1

WEST = 2

NORTH = 3

class ShipPosition(*horizontal: int*, *vertical: int*, *orientation:* Orientation)

Represent the current ferry position.

Attributes:

horizontal	Position of the ship in horizontal direction
vertical	Position of the ship in vertical direction
orientation	Orientation of the ship

Methods:

__repr__()	Return repr(self).
__eq__(other)	Return self==value.
__init__(horizontal, vertical, orientation)	

horizontal: int

Position of the ship in horizontal direction

vertical: int

Position of the ship in vertical direction

orientation: *Orientation*

Orientation of the ship

__repr__() → str

Return repr(self).

__eq__(*other*)

Return self==value.

__init__(*horizontal: int*, *vertical: int*, *orientation:* Orientation) → None

parse_input(*puzzle_input: str*) → *Lines*

Split the puzzle input along the newlines.

Requires

- `re.fullmatch(r"[NSEWLRF][0-9]+(n[NSEWLRF][0-9]+)*", puzzle_input)`

Ensures

- `"n".join(result) == puzzle_input`

update_position(*current_position:* ShipPosition, *move: str*) → *ShipPosition*

Execute a single `move` on `current_position` and return a new position.

Requires

- ```
  not (move[0] == "L" or move[0] == "R")
  or int(move[1:]) in [0, 90, 180, 270, 360]
  ```

- `re.match(r"^[NSEWLRF][0-9]+$", move)`

solve(*puzzle_input: str*) → *ShipPosition*

Execute the instructions and return the final ship's position.

Requires

- ```
  re.match(
      r"^(([NSEWF][0-9]+)|([LR](0|90|180|270|360)))"
      r"(\n(([NSEWF][0-9]+)|([LR](0|90|180|270|360))))*\Z",
      puzzle_input,
  )
  ```

2.2.13 Day 13: Shuttle Search

Functions:

next_departure(bus_id, min_time)	Compute the next departure of `bus_id` leaving earliest at `min_time`.
find_departure(start_time, bus_ids)	Find the earliest bus to catch after `start_time`.
parse_input(lines)	Parse the input into (earliest departure time, bus IDs).
main()	Execute the main routine.

next_departure(*bus_id: int*, *min_time: int*) → int

Compute the next departure of `bus_id` leaving earliest at `min_time`.

Requires

- `bus_id > 0`
- `min_time >= 0`

Ensures

- `min_time == 0 result == 0`

  (All buses leave at time zero.)

- `result < min_time + bus_id`
- `result >= min_time`

find_departure(*start_time: int*, *bus_ids: Set[int]*) → Tuple[int, int]

Find the earliest bus to catch after `start_time`.

Requires

- `all(i > 0 for i in bus_ids)`
- `len(bus_ids) > 0`
- `start_time >= 0`

Ensures

- `result[0] % result[1] == 0`

  (Departure time matches at least one bus.)

- `result[0] >= start_time`

parse_input(*lines:* Lines) → Tuple[int, Set[int]]

Parse the input into (earliest departure time, bus IDs).

Requires

- ```
  len(lines) == 2
  and re.match(r"\d+", lines[0])
  and re.match(r"(\d+|x)(,(\d+|x))*", lines[1])
  ```

- `len(lines) == 2`

main() → None

Execute the main routine.
  ```

2.2.14 Day 14: Docking Data

Classes:

Mask(clearing, setting)	Represent the bitmask of the initialization program.
Write(address, value)	Represent a write to the memory.
Program(mask, writes)	Represent the initialization program.
Memory(slots)	Represent the state of the memory.

Functions:

parse_mask(text)	Parse the text as an clearing and a setting mask, respectively.
parse_write(text)	Parse the write instruction and return (address, value).
parse_lines(lines)	Parse the input into an initialization program.
execute(program)	Execute the program and return the memory values.
sum_memory(memory)	Sum the values in the memory slots.

class Mask(*clearing: int*, *setting: int*)

Represent the bitmask of the initialization program.

Methods:

__init__(clearing, setting)	Initialize with the given values.

Attributes:

clearing	mask of the bits to be cleared (AND'ed)
setting	mask of the bits to be set (OR'ed)

__init__(*clearing: int*, *setting: int*) → None

Initialize with the given values.

Requires

- `0 <= setting <= 2**36 - 1`
- `0 <= clearing <= 2**36 - 1`

clearing: Final[int]

mask of the bits to be cleared (AND'ed)

setting: Final[int]

mask of the bits to be set (OR'ed)

parse_mask(*text: str*) → *Mask*

Parse the text as an clearing and a setting mask, respectively.

Requires

- `MASK_RE.match(text)`

Ensures

- `0 <= result.setting <= 2**36 - 1`

  (The setting mask not too large)

- `0 <= result.clearing <= 2**36 - 1`

  (The clearing mask not too large)

class Write(*address: int*, *value: int*)

Represent a write to the memory.

Methods:

__init__(address, value)	Initialize with the given values.

Attributes:

address	Address (offset) in the memory
value	Value to be written

__init__(*address: int*, *value: int*) → None

Initialize with the given values.

Requires

- `0 <= value <= 2**36 - 1`

  (The value in expected range)

- `address >= 0`

address: Final[int]

Address (offset) in the memory

value: Final[int]

Value to be written

parse_write(*text: str*) → Tuple[int, int]

Parse the write instruction and return (address, value).

Requires

- `WRITE_RE.match(text)`

Ensures

- `result[1] >= 0`

  (Value non-negative)

- `result[0] >= 0`

  (Address non-negative)

class Program(*mask:* Mask, *writes: List[*Write*]*)

Represent the initialization program.

Methods:

__init__(mask, writes)	Initialize with the given values.

Attributes:

mask	Mask used throughout the program
writes	Write instructions

__init__(*mask:* Mask, *writes: List[*Write*]*) → None

Initialize with the given values.

mask: Final[*Mask*]

Mask used throughout the program

writes: Final[List[*Write*]]

Write instructions

parse_lines(*lines:* Lines) → *Program*

Parse the input into an initialization program.

Requires

- `len(lines) > 2`
- `MASK_RE.match(lines[0])`
- `all(WRITE_RE.match(line) for line in lines[1:])`

class Memory(*slots: Mapping[int, int]*)

Represent the state of the memory.

Methods:

__init__(slots)	Initialize with the given values.

Attributes:

slots	Slot map as address value

__init__(*slots: Mapping[int, int]*) → None

Initialize with the given values.

Requires

- `all(key >= 0 for key in slots.keys())`

  (Addresses non-negative)

- `all(value >= 0 for value in slots.values())`

  (Values non-negative)

slots: Final[Mapping[int, int]]

Slot map as address value

execute(*program:* Program) → *Memory*

Execute the program and return the memory values.

sum_memory(*memory:* Memory) → int

Sum the values in the memory slots.

Ensures

- `result >= 0`

2.2.15 Day 15: Rambunctious Recitation

Data:

LAST_STEP	The last relevant step of the game

Functions:

solve(starting_numbers)	Play the memory game starting with the `starting_numbers` to``LAST_STEP``.

LAST_STEP = 2020

The last relevant step of the game

solve(*starting_numbers: List[int]*) → int

Play the memory game starting with the `starting_numbers` to``LAST_STEP``.

Requires

- `len(starting_numbers) > 0`

Ensures

- `0 <= result <= LAST_STEP - 2`

2.2.16 Day 16: Ticket Translation

Classes:

RuleParsing(identifier, ranges)	Represent a rule which is not constrained by pre-conditions.
Rule(identifier, ranges)	Represent a rule for the ticket field.

Functions:

applies(rule, value)	Check whether the `rule` applies to the `value`.
parse_rules(lines)	Parse the `lines` into rules.
parse_nearby_tickets(lines)	Parse the nearby tickets from `lines` to list of field values.
invalid_fields(rules, ticket)	Select the invalid fields from a `ticket` according to `rules`.
list_all_invalid_values(rules, tickets)	Select the invalid fields accross all `tickets` according to `rules`.
compute_error_rate(invalid_values)	Compute the error rate as sum of the invalid values.

class RuleParsing(*identifier: str*, *ranges: List[Tuple[int, int]]*)

Represent a rule which is not constrained by pre-conditions.

Methods:

__init__(identifier, ranges)	Initialize with the given values.

Attributes:

identifier	Identifier of the field
ranges	Valid range of values for the field

__init__(*identifier: str*, *ranges: List[Tuple[int, int]]*) → None

Initialize with the given values.

identifier: Final[str]

Identifier of the field

ranges: Final[List[Tuple[int, int]]]

Valid range of values for the field

class Rule(*identifier: str*, *ranges: List[Tuple[int, int]]*)

Represent a rule for the ticket field.

Methods:

__init__(identifier, ranges)	Initialize with the given values.

Attributes:

identifier	identifier of the field
ranges	acceptable ranges for the field

__init__(*identifier: str*, *ranges: List[Tuple[int, int]]*) → None

Initialize with the given values.

Requires

- `all(range[0] < range[1] for range in ranges)`
- `len(identifier) > 0`

identifier: Final[str]

identifier of the field

ranges: Final[List[Tuple[int, int]]]

acceptable ranges for the field

applies(*rule:* Rule, *value: int*) → bool

Check whether the `rule` applies to the `value`.

parse_rules(*lines:* Lines) → List[*RuleParsing*]

Parse the `lines` into rules.

While the parsed rules are syntactically correct, they have to be yet semantically verified.

Requires

- `all(RULE_RE.match(line) for line in lines)`

parse_nearby_tickets(*lines: List[str]*) → List[List[int]]

Parse the nearby tickets from `lines` to list of field values.

Requires

- ```
  all(
      re.match(r'^(0|[1-9][0-9]*)(,(0|[1-9][0-9]*))+\Z', line)
      for line in lines
  )
  ```

invalid_fields(*rules: List[*Rule*]*, *ticket: List[int]*) → List[int]

Select the invalid fields from a `ticket` according to `rules`.

Ensures

- ```
  all(
      value in ticket
      for value in result
  )
  ```

list_all_invalid_values(*rules: List[*Rule*]*, *tickets: List[List[int]]*) → List[int]

Select the invalid fields accross all `tickets` according to `rules`.

Ensures

- ```
  all(
      any(value in ticket for ticket in tickets)
      for value in result
  )
  ```

compute_error_rate(*invalid_values: List[int]*) → int

Compute the error rate as sum of the invalid values.

2.2.17 Day 17: Conway Cubes

Classes:

Point(x, y, z)	Represent a cube in 3D.
Activity(active_cubes)	Represent the current active cubes in the energy source

Functions:

are_neighbours(point, another)	Check whether the `point` and `another` are adjacent cubes.
list_neighbourhood(point)	List all neighbouring cubes w.r.t.
apply(activity)	Perform a single cycle of the initialization starting from `activity`.
parse_initial(lines)	Parse `lines` into the state of the energy source.
repr_activity(activity)	Represent the activity as a string for easier testing/debugging.
count_active(activity)	Count number of active cells in the `activity`.

class Point(*x: int*, *y: int*, *z: int*)

Represent a cube in 3D.

Methods:
  ```

__new__(cls, x, y, z)	Create the point as tuple with the given coordinates.
__getitem__(index)	Get the item at the given integer index.

static __new__(*cls*, *x: int*, *y: int*, *z: int*) → *Point*

Create the point as tuple with the given coordinates.

__getitem__(*index: int*) → int

Get the item at the given integer index.

class Activity(*active_cubes: Set[*Point*]*)

Represent the current active cubes in the energy source

Methods:

__new__(cls, active_cubes)	Create an activity as a set of active cubes.
__iter__()	Iterate over the lines.
__contains__(item)	Return True if the cube is activated.
__len__()	Return the number of active cubes.

static __new__(*cls*, *active_cubes: Set[*Point*]*) → *Activity*

Create an activity as a set of active cubes.

__iter__() → Iterator[*Point*]

Iterate over the lines.

__contains__(*item:* Point) → bool

Return True if the cube is activated.

__len__() → int

Return the number of active cubes.

are_neighbours(*point:* Point, *another:* Point) → bool

Check whether the `point` and `another` are adjacent cubes.

list_neighbourhood(*point:* Point) → List[*Point*]

List all neighbouring cubes w.r.t. the `point` cube.

Ensures

- ```
  all(
      are_neighbours(point=point, another=neighbour)
      for neighbour in result
  )
  ```
- `point not in result`
- `len(result) == 26`

apply(*activity:* Activity) → *Activity*

Perform a single cycle of the initialization starting from `activity`.

Returns

the new state of the energy source
  ```

parse_initial(*lines:* Lines) → *Activity*

Parse `lines` into the state of the energy source.

Requires

- `all(re.match(r'^[.#]+Z', line) for line in lines)`
- ```
  len(lines) >= 1
  and all(
      len(line) == len(lines[0])
      for line in lines
  )
  ```

Ensures

- `sum(line.count('#') for line in lines) == len(result)`

repr_activity(*activity:* Activity) → str

Represent the activity as a string for easier testing/debugging.

count_active(*activity:* Activity) → int

Count number of active cells in the `activity`.

Ensures

- `result >= 0`

2.2.18 Day 18: Operation Order

Classes:

Operation(value)	Represent a mathematical operation.
Tail(op, right)	Represent the tail of an expression.
Node(head, tail)	Represent a node of the abstract syntax tree of a mathematical expression.

Functions:

extract_expression(expr)	Extract the sub-expression surrounded by parentheses in `expr`.
parse(expression)	Parse the `expression` into an abstract syntax tree.
serialize(node)	Serialize the abstraction syntax tree given as `node` to a string.
compute(node)	Evaluate the parsed expression given as `node`.

class Operation(*value*)

Represent a mathematical operation.

Attributes:

ADD	Addition
MUL	Multiplication
  ```

ADD = '+'

Addition

MUL = '*'

Multiplication

class Tail(*op: Operation*, *right: Union[int, Node]*)

Represent the tail of an expression.

Attributes:

op	Operation of the mathematical expression
right	Right-hand side of the expression

Methods:

__delattr__(name)	Implement delattr(self, name).
__eq__(other)	Return self==value.
__init__(op, right)	
__repr__()	Return repr(self).
__setattr__(name, value)	Implement setattr(self, name, value).

op: *Operation*

Operation of the mathematical expression

right: Union[int, *Node*]

Right-hand side of the expression

__delattr__(*name*)

Implement delattr(self, name).

__eq__(*other*)

Return self==value.

__init__(*op: Operation*, *right: Union[int, Node]*) → None

__repr__()

Return repr(self).

__setattr__(*name*, *value*)

Implement setattr(self, name, value).

class Node(*head: Union[int, Node]*, *tail: List[Tail]*)

Represent a node of the abstract syntax tree of a mathematical expression.

Attributes:

head	Constant or left-hand side expression
tail	Remaining expressions on the right-hand side

Methods:

__delattr__(name)	Implement delattr(self, name).
__eq__(other)	Return self==value.
__init__(head, tail)	
__repr__()	Return repr(self).
__setattr__(name, value)	Implement setattr(self, name, value).

head: Union[int, *Node*]

Constant or left-hand side expression

tail: List[*Tail*]

Remaining expressions on the right-hand side

__delattr__(*name*)

Implement delattr(self, name).

__eq__(*other*)

Return self==value.

__init__(*head: Union[int,* Node*]*, *tail: List[*Tail*]*) → None

__repr__()

Return repr(self).

__setattr__(*name*, *value*)

Implement setattr(self, name, value).

extract_expression(*expr: str*) → str

Extract the sub-expression surrounded by parentheses in `expr`.

Requires

- `expr.count("(") == expr.count(")")`
- `re.match(r"^(.+)", expr)`

Ensures

- `result in expr`

parse(*expression: str*) → Optional[*Node*]

Parse the `expression` into an abstract syntax tree.

serialize(*node:* Node) → str

Serialize the abstraction syntax tree given as `node` to a string.

Ensures

- `parse(result) == node`

compute(*node:* Node) → int

Evaluate the parsed expression given as `node`.

2.2.19 Day 19: Monster Messages

Classes:

Rule()	Represent a rule that valid messages should obey.
RuleOr(rules)	Represent an union composition of rules (at least one rule matches).
RuleSequence(rules)	Represent a chain of rules where all rules need to match in sequence.
RuleLiteral(literal)	Represent a rule where a literal is exactly matched.
Node()	Represent a node in the abstract syntax tree.
NodeLiteral(literal)	Represent a literal in the rule syntax.
NodeReference(identifier)	Represent a reference to a rule.
NodeSequence(references)	Represent a parsed sequence of node references.
NodeOr(sequences)	Represent a parsed union of node sequences.

Functions:

parse_rule(line)	Parse the rule from the `line` into (rule identifier, abstract syntax tree).
parse_rules(lines)	Parse the rules from `lines` into a dictionary of identifier AST.
iterate(rule_tree)	Yield the `rule_tree` and all its descendants.
repr_rule_tree(rule_tree)	Represent the abstract syntax tree `rule_tree` as a string for inspection.
interpret_rule_0(rule_trees)	Interpret the rule trees and construct the rule 0.
count_matching_messages(rule_0, messages)	Count the `messages` that match the rules starting from `rule_0`.

class Rule

Represent a rule that valid messages should obey.

Methods:

match(text)	Match the `text` and return the remaining unmatched text.

abstract match(*text: str*) → Optional[str]

Match the `text` and return the remaining unmatched text.

If the beginning of the `text` does not match, return `None`.

Ensures

- `result is not None text.endswith(result)`
- ```
  not (result is not None)
  or (len(text) > len(result))
  ```

class RuleOr(*rules: List[Rule]*)

Represent an union composition of rules (at least one rule matches).

Methods:
  ```

__init__(rules)	Initialize with the given values.
match(text)	Check whether at least one rule from *rules* matches.

Attributes:

rules	Union of rules

__init__(*rules: List[*Rule*]*) → None

Initialize with the given values.

rules: Final[List[*Rule*]]

Union of rules

match(*text: str*) → Optional[str]

Check whether at least one rule from *rules* matches.

Ensures

- `result is not None text.endswith(result)`
- ```
  not (result is not None)
  or (len(text) > len(result))
  ```

class RuleSequence(*rules: List[*Rule*]*)

Represent a chain of rules where all rules need to match in sequence.

Methods:

__init__(rules)	Initialize with the given values.
match(text)	Check whether `text` matches the whole sequence of *rules*.

Attributes:

rules	Rule chain

__init__(*rules: List[*Rule*]*) → None

Initialize with the given values.

rules: Final[List[*Rule*]]

Rule chain

match(*text: str*) → Optional[str]

Check whether `text` matches the whole sequence of *rules*.

Ensures

- `result is not None text.endswith(result)`
- ```
  not (result is not None)
  or (len(text) > len(result))
  ```

class RuleLiteral(*literal: str*)

Represent a rule where a literal is exactly matched.

Methods:

__init__(literal)	Initialize with the given values.
match(text)	Check whether `text` matches exactly the *literal*.

Attributes:

literal	Literal to be matched

__init__(*literal: str*) → None

Initialize with the given values.

literal: Final[str]

Literal to be matched

match(*text: str*) → Optional[str]

Check whether `text` matches exactly the *literal*.

Ensures

- `result is not None text.endswith(result)`
- ```
  not (result is not None)
  or (len(text) > len(result))
  ```
- `result is not None self.literal + result == text`

class Node

Represent a node in the abstract syntax tree.

class NodeLiteral(*literal: str*)

Represent a literal in the rule syntax.

Methods:

__init__(literal)	Initialize with the given values.

Attributes:

literal	Parsed literal

__init__(*literal: str*) → None

Initialize with the given values.

literal: Final[str]

Parsed literal

class NodeReference(*identifier: int*)

Represent a reference to a rule.

Methods:

__init__(identifier)	Initialize with the given values.
  ```

Attributes:

identifier	Identifier of the referenced rule

__init__(*identifier: int*) → None

Initialize with the given values.

identifier: Final[int]

Identifier of the referenced rule

class NodeSequence(*references: List[*NodeReference*]*)

Represent a parsed sequence of node references.

Methods:

__init__(references)	Initialize with the given values.

Attributes:

references	Parsed references

__init__(*references: List[*NodeReference*]*) → None

Initialize with the given values.

references: Final[List[*NodeReference*]]

Parsed references

class NodeOr(*sequences: List[*NodeSequence*]*)

Represent a parsed union of node sequences.

Methods:

__init__(sequences)	Initialize with the given values.

Attributes:

sequences	Union

__init__(*sequences: List[*NodeSequence*]*) → None

Initialize with the given values.

sequences: Final[List[*NodeSequence*]]

Union

parse_rule(*line: str*) → Tuple[int, *Node*]

Parse the rule from the `line` into (rule identifier, abstract syntax tree).

Requires

- `RULE_RE.match(line)`

parse_rules(*lines:* Lines) → MutableMapping[int, *Node*]

Parse the rules from `lines` into a dictionary of identifier AST.

Requires

- `all(RULE_RE.match(line) for line in lines)`

iterate(*rule_tree:* Node) → Iterator[*Node*]

Yield the `rule_tree` and all its descendants.

repr_rule_tree(*rule_tree:* Node) → str

Represent the abstract syntax tree `rule_tree` as a string for inspection.

interpret_rule_0(*rule_trees: Mapping[int,* Node*]*) → *Rule*

Interpret the rule trees and construct the rule 0.

Requires

- ```
  all(
      all(
          node.identifier in rule_trees
          for node in iterate(rule_tree)
          if isinstance(node, NodeReference)
      )
      for rule_tree in rule_trees.values()
  )
  ```

  (No dangling references)

- `0 in rule_trees`

  (The initial rule is present.)

count_matching_messages(*rule_0:* Rule, *messages: List[str]*) → int

Count the `messages` that match the rules starting from `rule_0`.

Ensures

- `0 <= result <= len(messages)`

2.2.20 Day 20: Jurassic Jigsaw

Data:

VALID_SIDE_RE	Express the edge of a tile

Functions:
  ```

reverse_side(side)	Flip the side.
transform_tile(tile)	Produce the tile transformations by rotating and flipping it.
place_remaining_tiles(image, tiles)	Try to assemble the remaining tiles into the image.
place_tiles(tiles)	Assemble the tiles given as ID tile transformations into an image.
parse_tile(lines)	Parse the `lines` into (ID number, tile
parse_tiles(text)	Parse the input `text` into ID number possible tile transformations.
main()	Execute the main routine.

Classes:

Tile(top, right, bottom, left)	Represent a tile of the puzzle.
Image(width, tiles)	Represent a (partially or fully) assembled puzzle of tiles.
ValidTileText(lines)	Represent lines to conform to valid tile text.

VALID_SIDE_RE = re.compile('[.#]{10}')

Express the edge of a tile

reverse_side(*side: str*) → str

Flip the side.

Requires

- `VALID_SIDE_RE.fullmatch(side)`

Ensures

- `re.fullmatch(r"[.#]{10}", result)`

class Tile(*top: str*, *right: str*, *bottom: str*, *left: str*)

Represent a tile of the puzzle.

Methods:

__init__(top, right, bottom, left)	Initialize with the given values.
rotate()	Copy the tile and rotate it clock-wise.
flip_vertical()	Copy the tile and flip the it along the vertical axis.
flip_horizontal()	Copy the tile and flip it along the horizontal axis.
__repr__()	Represent the tile as string for easier debugging.
__eq__(other)	Compare by sides, if `other` is a *Tile*.

Attributes:

top	Top side
right	Right side
bottom	Bottom side
left	Left side

__init__(*top: str*, *right: str*, *bottom: str*, *left: str*) → None

Initialize with the given values.

Requires

- `left[-1] == top[0]`
- `bottom[-1] == left[0]`
- `right[-1] == bottom[0]`
- `top[-1] == right[0]`
- ```
  all(
      VALID_SIDE_RE.fullmatch(side)
      for side in (top, right, bottom, left)
  )
  ```

top: Final[str]

Top side

right: Final[str]

Right side

bottom: Final[str]

Bottom side

left: Final[str]

Left side

rotate() → *Tile*

Copy the tile and rotate it clock-wise.

flip_vertical() → *Tile*

Copy the tile and flip the it along the vertical axis.

flip_horizontal() → *Tile*

Copy the tile and flip it along the horizontal axis.

__repr__() → str

Represent the tile as string for easier debugging.

__eq__(*other: object*) → bool

Compare by sides, if `other` is a *Tile*.

Otherwise, by equality.

transform_tile(*tile:* Tile) → Set[*Tile*]

Produce the tile transformations by rotating and flipping it.

class Image(*width: int*, *tiles: List[Tuple[int,* Tile*]]*)

Represent a (partially or fully) assembled puzzle of tiles.

Attributes:

width	Total width of the image
tiles	Assembled tiles

Methods:
  ```

pop()	Remove the last tile from the puzzle.
attempt_add(tile_id, tile)	Try to add the tile into the image.
__eq__(other)	Return self==value.
__init__(width, tiles)	
__repr__()	Return repr(self).

width: int

Total width of the image

tiles: List[Tuple[int, *Tile*]]

Assembled tiles

pop() → Tuple[int, *Tile*]

Remove the last tile from the puzzle.

attempt_add(*tile_id: int*, *tile:* Tile) → bool

Try to add the tile into the image.

Returns
True if successful

__eq__(*other*)

Return self==value.

__init__(*width: int*, *tiles: List[Tuple[int,* Tile*]]*) → None

__repr__()

Return repr(self).

place_remaining_tiles(*image:* Image, *tiles: Dict[int, Set[*Tile*]]*) → bool

Try to assemble the remaining tiles into the image.

Returns
True if there are no more tiles left, or if the assembly was possible.

place_tiles(*tiles: Dict[int, Set[*Tile*]]*) → Optional[*Image*]

Assemble the tiles given as ID tile transformations into an image.

Returns
Image, if possible; None if no puzzle could be assembled

Requires

- `int(math.sqrt(len(tiles))) ** 2 == len(tiles)`

  (Number of tiles must be a perfect square)

class ValidTileText(*lines: Sequence[str]*)

Represent lines to conform to valid tile text.

Methods:

__new__(cls, lines)	Ensure the properties on the `lines`.
__getitem__()	Get the line(s) at the given index.
__len__()	Return the number of the lines.
__iter__()	Iterate over the lines.

static __new__(*cls*, *lines: Sequence[str]*) → *ValidTileText*

Ensure the properties on the `lines`.

Requires

- ```
  len(lines) == 11
  and re.match(r"Tile (\d+)", lines[0]) is not None
  and all(VALID_SIDE_RE.fullmatch(line) for line in lines[1:])
  ```

  (Raise `ValueError`)

__getitem__(*index: int*) → str

__getitem__(*index: slice*) → *ValidTileText*

Get the line(s) at the given index.

__len__() → int

Return the number of the lines.

__iter__() → Iterator[str]

Iterate over the lines.

parse_tile(*lines:* ValidTileText) → Tuple[int, *Tile*]

Parse the `lines` into (ID number, tile

parse_tiles(*text: str*) → Dict[int, Set[*Tile*]]

Parse the input `text` into ID number possible tile transformations.

main() → None

Execute the main routine.

2.2.21 Day 21: Allergen Assessment

Classes:

Identifier(value)	Represent an ingredient or allergen identifier.
Allergen(value)	Represent an identifier of an allergen.
Ingredient(value)	Represent an identifier of an ingredient.
IngredientLine(line)	Specify a well-formed line representing an ingredient list with the allergen.
Entry(ingredients, allergens)	Represent an entry in the list of foods.

Data:

INGREDIENT_LINE_RE	Express a line specifying an ingredient list along with the allergens.

Functions:

parse_ingredient_line(line)	Encapsulate the parsing of `line` into entries.
serialize_entry(entry)	Serialize the `entry` back into the string.
find_non_allergenic_ingredients(entries)	Find the ingredients without allergens.
solve(lines)	Parse the input and return the set of ingredients without allergens.
  ```

class Identifier(*value: str*)

Represent an ingredient or allergen identifier.

Methods:

__new__(cls, value)	Enforce the properties on the `identifier`.

static __new__(*cls*, *value: str*) → *Identifier*

Enforce the properties on the `identifier`.

Requires

- `re.fullmatch(r"^[a-zA-Z]+Z", value)`

class Allergen(*value: str*)

Represent an identifier of an allergen.

class Ingredient(*value: str*)

Represent an identifier of an ingredient.

INGREDIENT_LINE_RE = re.compile('\\s*(?P<ingredients>[a-zA-Z]+(\\s+[a-zA-Z]+)*)\\s+\\(contains\\s+(?P<allergens>[a-zA-Z]+(\\s*,\\s*[a-zA-Z]+)*)\\)')

Express a line specifying an ingredient list along with the allergens.

class IngredientLine(*line: str*)

Specify a well-formed line representing an ingredient list with the allergen.

Methods:

__new__(cls, line)	**requires**

static __new__(*cls*, *line: str*) → *IngredientLine*

Requires

- `INGREDIENT_LINE_RE.fullmatch(line)`

class Entry(*ingredients: List[*Ingredient*]*, *allergens: List[*Allergen*]*)

Represent an entry in the list of foods.

Methods:

__init__(ingredients, allergens)	Initialize with the given values.

Attributes:

ingredients	Ingredients of the entry
allergens	Allergens of the entry

__init__(*ingredients: List[*Ingredient*]*, *allergens: List[*Allergen*]*) → None

Initialize with the given values.

Requires

- `len(ingredients) > 0`

- `len(allergens) > 0`

ingredients: List[*Ingredient*]

Ingredients of the entry

allergens: List[*Allergen*]

Allergens of the entry

parse_ingredient_line(*line:* IngredientLine) → *Entry*

Encapsulate the parsing of `line` into entries.

serialize_entry(*entry:* Entry) → *IngredientLine*

Serialize the `entry` back into the string.

Ensures

- `result == serialize_entry(parse_ingredient_line(result))`

find_non_allergenic_ingredients(*entries: List[*Entry*]*) → Set[*Ingredient*]

Find the ingredients without allergens.

solve(*lines: List[*IngredientLine*]*) → Set[*Ingredient*]

Parse the input and return the set of ingredients without allergens.

2.2.22 Day 22: Crab Combat

Classes:

Deck(cards)	Represent a deck of cards.
Split(deck1, deck2)	Represent a split of one big deck of cards into two sub-decks.

Functions:

play_a_round(split)	Play a round of the game given the current `split`.
parse_lines(lines)	Parse the input lines into two decks, as list of cards.
compute_score(deck)	Compute the score for the given deck based on its cards.
play(split)	Play the game starting with the `split` until one of the players wins.

class Deck(*cards: Sequence[int]*)

Represent a deck of cards.

Please make sure that you transfer the "ownership" immediately to `Deck` and don't modify the original list of strings any more:

```
##
# OK
##

deck = Deck([1, 2, 3])

##
# Not OK
```

(continues on next page)

(continued from previous page)

```
##

cards = [1, 2, 3]
deck = Deck(cards)
# ... do something assuming ``deck`` is immutable ...

cards[0] = 2
# ERROR! cards[0] now breaks the invariant!
```

Methods:

__init__(cards)	Initialize with the given values.
__getitem__()	Get the card(s) at the given index.
__len__()	Return the number of the cards in the deck.
__iter__()	Iterate through the cards in the deck.
__add__(other)	Join two decks together.
__repr__()	Represent the deck for easier debugging.
__eq__(other)	Compare with `other` by *cards*.

Attributes:

cards	Cards in the deck

__init__(*cards: Sequence[int]*) → None

Initialize with the given values.

Requires

- `len(set(cards)) == len(cards)`

  (Unique cards)

- `all(card >= 0 for card in cards)`

cards: Final[Sequence[int]]

Cards in the deck

__getitem__(*index: int*) → int

__getitem__(*index: slice*) → *Deck*

Get the card(s) at the given index.

__len__() → int

Return the number of the cards in the deck.

__iter__() → Iterator[int]

Iterate through the cards in the deck.

__add__(*other:* Deck) → *Deck*

Join two decks together.

Requires

- ```
  (
          sum := list(self.cards) + other.cards,
          len(set(sum)) == len(sum)
  )[1]
  ```

  (Unique cards after the addition)

__repr__() → str

Represent the deck for easier debugging.

__eq__(*other: object*) → bool

Compare with `other` by *cards*.

If `other` is not a *Deck* or `List:, propagate to generic ``__eq__`.

class Split(*deck1:* Deck, *deck2:* Deck)

Represent a split of one big deck of cards into two sub-decks.

Methods:

__init__(deck1, deck2)	Initialize with the given values.

Attributes:

deck1	The deck for the player 1
deck2	The deck for the player 2

__init__(*deck1:* Deck, *deck2:* Deck) → None

Initialize with the given values.

Requires

- `not set(deck1).intersection(deck2)`

  (No overlapping cards)

deck1: Final[*Deck*]

The deck for the player 1

deck2: Final[*Deck*]

The deck for the player 2

play_a_round(*split:* Split) → *Split*

Play a round of the game given the current `split`.

Returns

A new split after the round

Requires

- `len(split.deck2) > 0`

  (Not game over for player 2)

- `len(split.deck1) > 0`

  (Not game over for player 1)

Ensures
  ```

- ```
  (
          len(split.deck1) == len(result.deck1) + 1
          and len(split.deck2) == len(result.deck2) - 1)
  or (
          len(split.deck1) == len(result.deck1) - 1
          and len(split.deck2) == len(result.deck2) + 1)
  ```

  (Either lost or won two cards)

- `split.deck2[1:]  == result.deck2[0:len(split.deck2) - 1]`

  (Only the prefix and the suffix of the deck 2 change)

- `split.deck1[1:]  == result.deck1[0:len(split.deck1) - 1]`

  (Only the prefix and the suffix of the deck 1 change)

- `set(split.deck1).union(split.deck2) == set(result.deck1).union(result.deck2)`

  (No new cards)

parse_lines(*lines: List[str]*) → Tuple[List[int], List[int]]

Parse the input lines into two decks, as list of cards.

Requires

- `len(lines) > 3`
- `lines[0] == 'Player 1:'`
- `'Player 2:'  in lines[1:]`
- ```
  all(
      re.match(r'^(Player 1:|Player 2:|0|[1-9][0-9]*|)\Z', line)
      for line in lines
  )
  ```

compute_score(*deck:* Deck) → int

Compute the score for the given deck based on its cards.

Ensures

- `result >= 0`

play(*split:* Split) → *Split*

Play the game starting with the `split` until one of the players wins.

Requires

- `len(split.deck2) > 0`

  (Not game over for player 2)

- `len(split.deck1) > 0`

  (Not game over for player 1)

Ensures

- `set(split.deck1).union(split.deck2) == set(result.deck1).union(result.deck2)`

- ```
  (
          len(split.deck1) + len(split.deck2) == len(result.deck1)
          and len(result.deck2) == 0
  )
  or (
          len(result.deck1) == 0
          and len(split.deck1) + len(split.deck2) == len(result.deck2)
  )
  ```

2.2.23 Day 23: Crab Cups

Classes:

Cup(label[, next_cup])	Represent a cup with a label and the cup next to it clockwise.
CupCircle()	Represent a circle of cups as a circular linked list.

Functions:

cup_circle_to_str(cup_circle)	Stringify the labels of the `cup_circle`, starting from the current cup.
initialize_cups(cup_labels)	Create a CupCircle from `cup_labels`.
crab_move(cup_circle)	Perform one move by the crab.
solve_100_steps(cup_labels)	Solve the problem for 100 crab moves.

class Cup(*label: int*, *next_cup: Optional[Cup] = None*)

Represent a cup with a label and the cup next to it clockwise.

Methods:

__init__(label[, next_cup])	Initialize with the given values.

Attributes:

label	label of the cup
next_cup	the next cup clockwise

__init__(*label: int*, *next_cup: Optional[Cup] = None*) → None

Initialize with the given values.

label: int

label of the cup

next_cup: *Cup*

the next cup clockwise

class CupCircle

Represent a circle of cups as a circular linked list.

Methods:
  ```

__init__()	Initialize the circle as an empty circular linked list.
add_new_cup(label)	Add a new cup to the circle with `label`.
__repr__()	Represent the circle as its labels.
__eq__(other)	Return whether two circles are identical in the labels of the circle.
__len__()	Return the number of cups in the circle.

Attributes:

current_cup	the cup from which each new move starts

__init__() → None

Initialize the circle as an empty circular linked list.

current_cup: Optional[*Cup*]

the cup from which each new move starts

add_new_cup(*label: int*) → None

Add a new cup to the circle with `label`.

The cup is added next to the current cup counter-clockwise.

Requires

- `label >= 0`
- `not self._is_label_in_circle(label)`

__repr__() → str

Represent the circle as its labels.

__eq__(*other: object*) → bool

Return whether two circles are identical in the labels of the circle.

__len__() → int

Return the number of cups in the circle.

cup_circle_to_str(*cup_circle:* CupCircle) → str

Stringify the labels of the `cup_circle`, starting from the current cup.

Ensures

- `cup_circle == initialize_cups(result)`

initialize_cups(*cup_labels: str*) → *CupCircle*

Create a CupCircle from `cup_labels`.

Requires

- `len(set(cup_labels)) == len(cup_labels)`
- `NUMBER_RE.fullmatch(cup_labels)`

Ensures

- `cup_labels == cup_circle_to_str(result)`

`crab_move`(*cup_circle:* CupCircle) → None

Perform one move by the crab.

Requires

- `len(cup_circle) >= 5`

OLD

- `.len_cup_circle` = `len(cup_circle)`

Ensures

- `OLD.len_cup_circle == len(cup_circle)`

`solve_100_steps`(*cup_labels: str*) → *CupCircle*

Solve the problem for 100 crab moves.

Requires

- `len(set(cup_labels)) == len(cup_labels)`
- `NUMBER_RE.fullmatch(cup_labels)`
- `len(cup_labels) >= 5`

Ensures

- `len(cup_labels) == len(result)`

2.2.24 Day 24: Lobby Layout

Classes:

Cell(x, y, z)	Represent a hexagonal cell of the layout using cube coordinates.
Direction(value)	Enumerate the possible directions from a hexagonal cell.

Functions:

cell_as_tuple(cell)	Convert the `cell` into a tuple of x, y and z coordinates.
next_cell(cell, direction)	Retrieve the next cell starting from `cell` in the `direction`.
follow_directions(start, directions)	Walk the `directions` from the `start`.
parse_line(line)	Parse the input line.
stringify_directions(directions)	Represent the directions as a concatenated text.
count_flips(plan)	Count how many cells had to flip for the given `plan`.

Data:

VALUE_TO_DIRECTION	Map string literal of *Direction* *Direction*.
DIRECTIONS_RE	Express a list of directions
ONE_DIRECTION_RE	Express a single direction

class Cell(*x: int*, *y: int*, *z: int*)

Represent a hexagonal cell of the layout using cube coordinates.

See https://www.redblobgames.com/grids/hexagons/.

Methods:

__init__(x, y, z)	Initialize with the given values.

Attributes:

x	X-coordinate (south-east, north-west)
y	Y-coordinate (south-west, north-east)
z	Z-coordinate (east, west)

__init__(*x: int*, *y: int*, *z: int*) → None

Initialize with the given values.

Requires

- `x + y + z == 0`

x: Final[int]

X-coordinate (south-east, north-west)

y: Final[int]

Y-coordinate (south-west, north-east)

z: Final[int]

Z-coordinate (east, west)

cell_as_tuple(*cell: Cell*) → Tuple[int, int, int]

Convert the `cell` into a tuple of x, y and z coordinates.

class Direction(*value*)

Enumerate the possible directions from a hexagonal cell.

Attributes:

EAST
SOUTH_EAST
SOUTH_WEST
WEST
NORTH_WEST
NORTH_EAST

EAST = 'e'

SOUTH_EAST = 'se'

```
    SOUTH_WEST = 'sw'

    WEST = 'w'

    NORTH_WEST = 'nw'

    NORTH_EAST = 'ne'

VALUE_TO_DIRECTION = {'e': Direction.EAST, 'ne': Direction.NORTH_EAST, 'nw':
Direction.NORTH_WEST, 'se': Direction.SOUTH_EAST, 'sw': Direction.SOUTH_WEST, 'w':
Direction.WEST}
```

Map string literal of *Direction Direction*.

next_cell(*cell:* Cell, *direction:* Direction) → *Cell*

Retrieve the next cell starting from `cell` in the `direction`.

follow_directions(*start:* Cell, *directions: List[*Direction*]*) → *Cell*

Walk the `directions` from the `start`.

Returns

The final *Cell* of the journey

```
DIRECTIONS_RE = re.compile('^(se|sw|nw|ne|w|e)+\\Z')
```

Express a list of directions

```
ONE_DIRECTION_RE = re.compile('(se|sw|nw|ne|w|e)')
```

Express a single direction

parse_line(*line: str*) → List[*Direction*]

Parse the input line.

Requires

- `len(line) > 0 DIRECTIONS_RE.match(line)`

Ensures

- `stringify_directions(result) == line`
- `len(line) == 0 len(result) == 0`

stringify_directions(*directions: List[*Direction*]*) → str

Represent the directions as a concatenated text.

Ensures

- `parse_line(result) == directions`

count_flips(*plan: List[List[*Direction*]]*) → int

Count how many cells had to flip for the given `plan`.

The `plan` consists of different journeys, all starting from the cell zero.

2.2.25 Day 25: Combo Breaker

Functions:

transform(subject, loop_size)	Transform the `subject` in `loop_size` steps of a hard-coded algorithm.
deduce_loop_size(subject, public_key)	Deduce the loop size for `public_key` by transforming the `subject`.
deduce_encryption_key(door_public_key, ...)	Figure out the subject number.

transform(*subject: int*, *loop_size: int*) → int

Transform the `subject` in `loop_size` steps of a hard-coded algorithm.

Requires

- `subject >= 0`
- `loop_size >= 0`

Ensures

- `0 <= result < 20201227`

deduce_loop_size(*subject: int*, *public_key: int*) → int

Deduce the loop size for `public_key` by transforming the `subject`.

Returns

The loop size, or -1 if no success

Requires

- `subject >= 0`

Ensures

- `result != -1 transform(subject, result) == public_key`

deduce_encryption_key(*door_public_key: int*, *card_public_key: int*) → int

Figure out the subject number.

Requires

- `card_public_key >= 0`
- `door_public_key >= 0`

2.3 Solutions to ETHZ "Introduction to Programming" 2019

We translated the exercises from the lecture "Introduction to Programming" ("Einführung in die Programmierung", 252-0027) held at ETH Zurich, Switzerland, as part of the Computer Science curriculum in Fall 2019.

You can find our auto-documented solutions here, while the source code is available at: https://github.com/mristin/python-by-contract-corpus/tree/main/python_by_contract_corpus/correct/ethz_eprog_2019

Please see *Details about the Exercises* why we chose only a subset of exercises and the rationale behind our choice.

2.3.1 Details about the Exercises

We picked only the subset of the exercises. The original exercises encompassed broader topics than programming (such as Extended Backus–Naur form, or short EBNF), so we skipped them. Additionally, we skipped the exercises such as the Exercise 0 which introduced the students to version control and usage of IDEs. As the course was meant to teach Java, some exercises focused on Java-specific topics. Since we are collecting a corpus of Python programs annotated with contracts, we skipped such exercises as we did not deem them relevant for the corpus. Some of the exercises involve programming of the graphical user interfaces (GUIs). While contracts are indeed very useful in the GUI programming, we consider GUI programming to be out-of-scope for the current corpus. Finally, we excluded the problems which we thought were too simple such as dealing with basic input/output operations (such as Exercise 2, Problem 4).

The original problem statements are much longer and more elaborate. As our focus is on the source code, we shortened the text as much as possible and give it here only as orientation about the complexity of the exercises. The problems themselves were sometimes simplified to make for a more pointed and easier-to-read code (*e.g.*, Exercise 6, Problem 5). We explicitly marked the corresponding changes in the description of the problem.

Please refer to the original text (in German) for more details about the exercises.

The original content of the exercises is courtesy of Laboratory for Software Technology, ETH Zurich, which also bears the copyright. A special thanks goes to Prof. Thomas Gross for pointing us to the exercises and sharing them with the public!

You can find the original exercises at: https://www.lst.inf.ethz.ch/education/archive/Fall2019/einfuehrung-in-die-programmierung-i–252-0027-.html

Reasons for Exclusions

We provide in the following the complete list of skipped problems and the reason for exclusion:

Exercise 0 and 1

Exercise 0 and Exercise 1 are introductory exercises about EBNF, how to use version control and IDE.

Exercise 2

- Problem 1 introduces general debugging practices in IDE.
- Problem 4 teaches console I/O which is too simple for contracts.

Exercise 4

- Problem 4 focuses on programming graphical user interfaces (GUIs) in Java.

Exercise 5

- Problem 1 concerns unit testing with JUnit.
- Problem 2 is too simple for the contracts.
- Problem 5 is a GUI exercise.

Exercise 6

- Problem 2 is Java-specific (introducing how to work with constructors and classes). In our opinion, it was an extension of Exercise 5, problem 3 which does not bring enough material in terms of contracts to merit the inclusion in the corpus.
- Problem 3 is about black-box testing with JUnit, hence no solution can be written in Python. However, it would be interesting in the future work to see how contracts can help black-box testing.

Exercise 7

- Problem 1 is almost identical to Exercise 6, Problem 4. Namely, a double-linked list needs to be implemented instead of a single-linked list. Since the interfaces and the contracts are the same, the solution to this problem would bring about nothing new in terms of contracts, so we decided to exclude it.
- Problem 3 is about EBNF and not a programming exercise.

Exercise 8

- Problem 4 concerns the black-box testing.

Exercise 9

- Problem 1 teaches the class hierarchy in Java, so we exclude it as irrelevant for the contracts.
- Problem 5 asks for implementation of a game. It would be really interesting to see how contracts help game development, but this is out-of-scope for the current corpus.

Exercise 10

- Problem 1 is focused on concepts of object-oriented programming such as interfaces and classes. The problem statements was very complex to motivate complex class hierarchies, but we found the solution to have a limited contribution in terms of contracts.
- Problem 2 is about drawing a map in the GUI. Similar to other GUI-related problems, we skip it and leave it for future work.
- Problem 3 makes the student practice exceptions, which is not relevant for our corpus.

Exercise 11

- Problem 3 explains how to draw functions in GUI.
- Problem 4 focuses on interfaces and their application. It would be a nice example for demonstrating inheritance of the contracts, but involves GUI programming, so we exclude it.

Exercise 12

- Problem 2 requires drawing the shapes on the screen. We exclude the problem as it is GUI-related.
- Problem 5 is not a programming problem (but concerns Hoare triples ;-)).

2.3.2 Solutions to Exercise 2

These are the auto-documented solutions to the Exercise 2 used during the lecture "Introduction to Programming" at ETH Zurich (Switzerland) in Fall 2019.

The text of the exercise is available (in German) at: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2019/0027_Intro/Homework/u2.pdf

Problem 2

Draw the following pattern:

```
  \/
 \\//
\\\///
///\\\
 //\\
  /\
```

You are given the size of the image (as width).

Data:

TRAILING_SPACE_RE	Express the trailing whitespace at the end of line

Functions:

draw(width)	Draw the pattern with size given as `width` and return the text lines.

`TRAILING_SPACE_RE = re.compile('\\s$')`

Express the trailing whitespace at the end of line

`draw`(*width: int*) → *Lines*

Draw the pattern with size given as `width` and return the text lines.

Requires

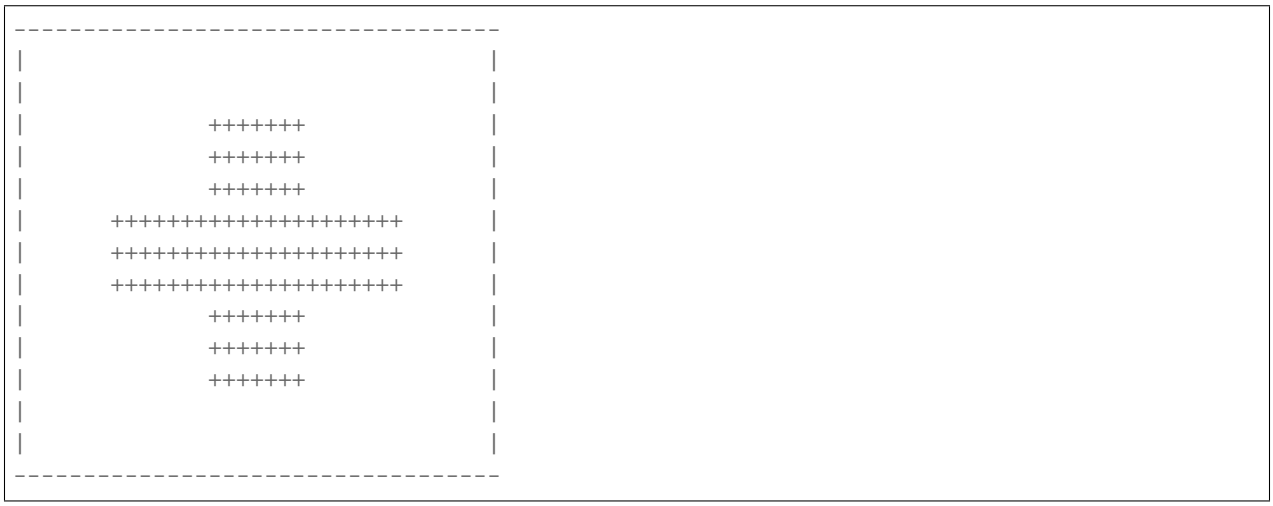
- `width % 2 == 0`
- `width > 0`

Ensures

- `all(len(line) > 0 for line in result)`
- `len(result) > 0`
- `len(result) % 2 == 0`
- `all(not TRAILING_SPACE_RE.match(line) for line in result)`
- ```
  all(
      len(line.strip()) % 2 == 0
      for line in result
  )
  ```
- ```
  all(
      line.strip().startswith('/') and line.endswith('\\')
      for line in result[int(len(result) / 2):]
  )
  ```
- ```
  all(
      line.strip().startswith('\\') and line.endswith('/')
      for line in result[:int(len(result) / 2)]
  )
  ```

Problem 3

Draw the Swiss flag:



(We extend the exercise a little bit so that it is a tidy bit more complex and fun to solve.) You are given the size of the image (as width and height).

Both the width and the height are multiples of 5.

Functions:

draw(width, height)	Draw the pattern of the size `width` x `height` and return the text lines.
  ```

draw(*width: int*, *height: int*) → *Lines*

Draw the pattern of the size `width` x `height` and return the text lines.

Requires

- `height % 5 == 0`
- `height > 0`
- `width % 5 == 0`
- `width > 0`

Ensures

- `len(result) > 0`
- `all(len(line) == width for line in result)`
- `len(result) == height`
- `ONLY_DASHES_RE.match(result[-1])`

  (bottom border)

- `ONLY_DASHES_RE.match(result[0])`

  (top border)

- ```
  all(
      line.startswith('|') and line.endswith('|')
      for line in result[1:-1]
  )
  ```

  (Vertical border)

- ```
  all(
      (
              center := len(line) // 2,
              line[:center] == line[center + 1:][::-1]
              if len(line) % 2 == 1
              else line[:center] == line[center:][::-1]
      )[1]
      for line in result
  )
  ```

  (Horizontal symmetry)

- ```
  all(
      result[i - 1] == result[len(result) - i]
      for i in range(1, int(len(result) / 2))
  )
  ```

  (Vertical symmetry)
  ```

Problem 5, Part 1

Draw the following pattern:

```
**..**..**..
..**..**..**
**..**..**..
..**..**..**
**..**..**..
..**..**..**
```

You are given the size of the image (as width).

Data:

ALLOWED_CHARS	Express every line of the pattern

Functions:

draw(width)	Draw the pattern with the size given as `width` and return the text lines.

ALLOWED_CHARS = re.compile('[.*]+')

Express every line of the pattern

draw(*width: int*) → *Lines*

Draw the pattern with the size given as `width` and return the text lines.

Requires

- `width % 4 == 0`
- `width > 0`

Ensures

- `len(result) == width / 2`
- `all(len(line) == width for line in result)`
- `result[0].endswith('.')`
- `result[0].startswith('*')`
- `result[-1].endswith('*')`
- `result[-1].startswith('.')`
- ```
  all(
      ALLOWED_CHARS.fullmatch(line)
      for line in result
  )
  ```
  ```

Problem 5, Part 2

Draw the following pattern:

```
....1....
...222...
..33333..
.4444444.
555555555
.6666666.
..77777..
...888...
....9....
```

You are given the size of the image (as height).

Data:

DIGITS_RE	Express a sequence of digits
PATTERN_RE	Express the output line

Functions:

draw(height)	Draw the pattern of size given as `height` and return the text lines.

`DIGITS_RE = re.compile('\\d+')`

Express a sequence of digits

`PATTERN_RE = re.compile('^(?P<leftpad>[.]*)\\d+(?P<rightpad>[.]*)$')`

Express the output line

`draw`(*height: int*) → *Lines*

Draw the pattern of size given as `height` and return the text lines.

Requires

- `height <= 9`
- `height % 2 == 1`
- `height > 0`

Ensures

- `len(result) == height`
- `all(len(line) == height for line in result)`
- ```
  all(
  (
          mtch := PATTERN_RE.match(line),
          mtch is not None
          and mtch.group('leftpad') == mtch.group('rightpad')
  )[1]
  for line in result)
  ```
  ```

- ```
  (
          middle := int(height / 2),
          DIGITS_RE.fullmatch(result[middle])
  )[1]
  ```

2.3.3 Solutions to Exercise 3

These are the auto-documented solutions to the Exercise 3 used during the lecture "Introduction to Programming" at ETH Zurich (Switzerland) in Fall 2019.

The text of the exercise is available (in German) at: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2019/0027_Intro/Homework/u3.pdf

Problem 1

For a given `n`, compute the sum:

```
s = 1 / (1**2) + 1 / (2**2) + ... + 1 / (n**2)
```

For `n == 0`, the sum should be 0.

For `n == 4`, the sum should be about 1.42.

Functions:

compute(n)	Compute the recursive sum `s = 1 / (1**2) + 1 / (2**2) + ... + 1 / (n**2)`.

compute(*n: int*) → float

Compute the recursive sum `s = 1 / (1**2) + 1 / (2**2) + ... + 1 / (n**2)`.

```
>>> compute(0)
0
```

```
>>> compute(4)
1.4236111111111112
```

Requires

- `n >= 0`

Ensures

- `result >= 0`
- `result < 2`
```

Problem 2

Give the binary representation of a non-zero positive integer `n`.

For example, for `n == 14`, the program should output `0b1110`.

Please do not use `bin(.)` built-in function.

Functions:

repr_binary(n)	Represent `n` as a binary number.

repr_binary(*n: int*) → str

Represent `n` as a binary number.

Requires

- `n > 0`

Ensures

- `bin(n) == result`

Problem 3

Compute a greatest common divisor (GCD) between two positive integers `x` and `y`.

Please note:

1) If `x` is greater-equal `y` and `x` is divisible by `y`, the GCD between `x` and `y` is `y`.
2) Otherwise, the GCD between `x` and `y` is `GCD(y, x % y)`.

For `x == 36` and `y == 44`, the GCD is 4.

Do not use `math.gcd` function.

Functions:

gcd(x, y)	Compute the greatest common divisor (GCD) between `x` and `y`.

gcd(*x: int*, *y: int*) → int

Compute the greatest common divisor (GCD) between `x` and `y`.

Requires

- `y > 0`
- `x > 0`

Ensures

- `result == math.gcd(x, y)`

Problem 4

Recognize numbers shown on a digital LED display.

The number is encoded as follows:

```
    a
   ___
f |   |
  | g | b
   ---
  |   | c
e |___|
    d
```

For example, the number 2 is encoded as `abged`.

Data:

VALID_LETTERS	Specify the valid identifiers for the segments.
TO_NUMBER	Define the mapping: segments identifier displayed number.
FROM_NUMBER	Define the inverse mapping of *TO_NUMBER*: number segments identifier

Functions:

decode(text)	Decode the given `text`, representing segment identifiers, as a digit.
encode(number)	Encode the given `number` as segments identifiers of the LED display.

VALID_LETTERS = {'a', 'b', 'c', 'd', 'e', 'f', 'g'}

Specify the valid identifiers for the segments.

TO_NUMBER = {'abc': 7, 'abcdef': 0, 'abcdefg': 8, 'abcdfg': 9, 'abcdg': 3, 'abdeg': 2, 'acdefg': 6, 'acdfg': 5, 'bc': 1, 'bcfg': 4}

Define the mapping: segments identifier displayed number.

FROM_NUMBER = {0: 'abcdef', 1: 'bc', 2: 'abdeg', 3: 'abcdg', 4: 'bcfg', 5: 'acdfg', 6: 'acdefg', 7: 'abc', 8: 'abcdefg', 9: 'abcdfg'}

Define the inverse mapping of *TO_NUMBER*: number segments identifier

decode(*text: str*) → int

Decode the given `text`, representing segment identifiers, as a digit.

Requires

- `all(letter in VALID_LETTERS for letter in text)`

  (only valid identifiers of line segments)

- `len(set(text)) == len(text)`

  (only one entry per line segment)

Ensures

- `0 <= result <= 9`

encode(*number: int*) → str

Encode the given `number` as segments identifiers of the LED display.

Requires

- `0 <= number <= 9`

Ensures

- `number == decode(result)`

2.3.4 Solutions to Exercise 4

These are the auto-documented solutions to the Exercise 4 used during the lecture "Introduction to Programming" at ETH Zurich (Switzerland) in Fall 2019.

The text of the exercise is available (in German) at: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2019/0027_Intro/Homework/u4.pdf

Problem 1

Compute the Sieve of Eratosthenes.

List all the prime numbers given a `limit` which are greater than 1 and smaller-equal `limit`.

Functions:

find(a, x)	Locate the leftmost value in `a` exactly equal to `x`.
naive_is_prime(number)	Check naively whether the `number` is a prime number.
sieve(limit)	Apply the Sieve of Eratosthenes on the numbers up to `limit`.

find(*a: List[int]*, *x: int*) → int

Locate the leftmost value in `a` exactly equal to `x`.

Return -1 if the element was not found in `a`.

Ensures

- `result != -1 0 <= result < len(a)`

naive_is_prime(*number: int*) → bool

Check naively whether the `number` is a prime number.

sieve(*limit: int*) → List[int]

Apply the Sieve of Eratosthenes on the numbers up to `limit`.

Returns

list of prime numbers till `limit`

Requires

- `limit > 1`

Ensures

- `len(result) == len(set(result))`

  (Unique results)

- ```
  all(
      1 < number <= limit
      for number in result
  )
  ```

- ```
  all(
      naive_is_prime(number)
      for number in result
  )
  ```

Problem 2

Approximate the square root of non-zero positive integer `c`.

Use Newton-Raphson method: `t' = ((c/t) +t) / 2.0`.

The result `t` should be precise up to `eps`: `abs(t*t -c) < eps`.

Functions:

approximate_sqrt(c, eps)	Approximate the square-root of `c` up to the precision `eps`.

approximate_sqrt(*c: int*, *eps: float*) → float

Approximate the square-root of `c` up to the precision `eps`.

Requires

- `eps > 1e-6`
- `c < 1000 * 1000`
- `c > 0`

Ensures

- `abs(result * result - c) <= eps`

  (Guaranteed precision)

Problem 3

Find two summands `s1` and `s2` for a given non-negative number `n`.

The following properties should hold: * `s1  s2` * `s1 + s2 == n` * The digit 7 does not appear neither in `s1` nor in `s2`.

Though the original problem did not state it, we enforce `s1 > 0` and `s2 > 0`.

Functions:

number_to_digits(n)	Disassemble the integer `n` into individual digits.
digits_to_number(digits)	Assemble the integer from the given `digits`.
find_summands(n)	Find the two summands `(s1, s2)` which satisfy the conditions.

number_to_digits(*n: int*) → List[int]

Disassemble the integer `n` into individual digits.

Requires

- `n > 0`

Ensures

- `digits_to_number(result) == n`
- `all(0 <= digit <= 9 for digit in result)`
- `result[0] != 0`
- `"".join(str(digit) for digit in result) == str(n)`

digits_to_number(*digits: Reversible[int]*) → int

Assemble the integer from the given `digits`.

Requires

- `all(0 <= digit <= 9 for digit in digits)`
- `len(digits) > 0`

find_summands(*n: int*) → Tuple[int, int]

Find the two summands `(s1, s2)` which satisfy the conditions.

- `s1  s2`
- `s1 + s2 == n`
- The digit 7 does not appear neither in `s1` nor in `s2`.

Requires

- `n > 2`

Ensures

- `"7" not in str(result[1])`
- `"7" not in str(result[0])`
- `result[0] >= result[1]`
- `result[1] > 0`
- `result[0] > 0`

2.3.5 Solutions to Exercise 5

These are the auto-documented solutions to the Exercise 5 used during the lecture "Introduction to Programming" at ETH Zurich (Switzerland) in Fall 2019.

The text of the exercise is available (in German) at: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2019/0027_Intro/Homework/u5.pdf

Problem 3

Analyze the height data of a population.

The data is available at: http://jse.amstat.org/v11n2/datasets.heinz.html.

Given a series of measurements in centimeters, compute: * The minimum, the maximum and average height of the population, * A histogram of the heights given the number of bins

Classes:

Measurement(value)	Represent a single measurement of a human height.
Range(start, end)	Represent a range of measurements.
BinRanges(bin_count, lower_bound, ...)	Represent the ranges of the histogram bins.
Histogram(ranges)	Represent a mutable histogram.

Functions:

compute_stats(measurements)	Compute the statistics of the given `measurements`.
bin_index(ranges, value)	Find the index of the bin range among `ranges` corresponding to `value`.
compute_histogram(measurements)	Compute the histogram over `measurements`.

class Measurement(*value: float*)

Represent a single measurement of a human height.

Methods:

__new__(cls, value)	Enforce the valid range on the measurement.

static __new__(*cls*, *value: float*) → *Measurement*

Enforce the valid range on the measurement.

Requires

- `0 < value < 400`

(Only valid value; the tallest man on earth ever measured was 251cm tall.)

compute_stats(*measurements: List[*Measurement*]*) → Tuple[float, float, float]

Compute the statistics of the given `measurements`.

Returns

Minimum, mean, maximum

Requires

- `len(measurements) > 0`

Ensures

```
not (len(set(measurements)) != 1)
or result[0] < result[1] < result[2]
```

```
not (len(set(measurements)) == 1)
or result[0] == result[1] == result[2]
```

(Identical measurements all give the same min, average, max)

class Range(*start: float*, *end: float*)

Represent a range of measurements.

Methods:

__init__(start, end)	Initialize with the given values.
__repr__()	Represent as mathematical range for easier debugging.

__init__(*start: float*, *end: float*) → None

Initialize with the given values.

Requires

- `not math.isnan(end)`
- `not math.isnan(start)`
- `start < end`

__repr__() → str

Represent as mathematical range for easier debugging.

class BinRanges(*bin_count: int*, *lower_bound: float*, *upper_bound: float*, *include_minus_inf: bool*, *include_inf: bool*)

Represent the ranges of the histogram bins.

Methods:

__new__(cls, bin_count, lower_bound, ...)	Construct `bin_count` number of histogram bins between `lower_bound` and `upper_bound`.
__getitem__()	Get the bin range at the given index.
__len__()	Return the number of the bin ranges.
__iter__()	Iterate over the bin ranges.

static __new__(*cls*, *bin_count: int*, *lower_bound: float*, *upper_bound: float*, *include_minus_inf: bool*, *include_inf: bool*) → *BinRanges*

Construct `bin_count` number of histogram bins between `lower_bound` and `upper_bound`.

If `include_inf`, include -∞ and +∞ in the spanned total range of histogram.

Requires

- ```
  (
          bin_width := (upper_bound - lower_bound) / bin_count,
          bin_width != 0
  )[1]
  ```

  (Bin width not numerically zero)

- `not math.isnan(lower_bound) and not math.isinf(lower_bound)`
- `not math.isnan(upper_bound) and not math.isinf(upper_bound)`
- `lower_bound < upper_bound`

Ensures
  ```

- `include_inf and include_minus_inf len(result) == bin_count + 2`

  (bin_count does not refer to +/- inf bins)
- `not include_inf and include_minus_inf len(result) == bin_count + 1`

  (bin_count does not refer to +/- inf bins)
- `include_inf and not include_minus_inf len(result) == bin_count + 1`

  (bin_count does not refer to +/- inf bins)
- `not include_inf and not include_minus_inf len(result) == bin_count`

  (bin_count does not refer to +/- inf bins)
- `not (include_inf ^ math.isinf(result[-1].end))`

  (include_inf <=> upper bound of the last bin is +inf)
- `not (include_minus_inf ^ math.isinf(result[0].start))`

  (include_min_inf <=> lower bound of the first bin is -inf)
- ```
  all(
      previous.end == current.start
      for previous, current in common.pairwise(result)
  )
  ```

  (Bin ranges without a hole)

__getitem__(*index: int*) → *Range*

__getitem__(*index: slice*) → *BinRanges*

Get the bin range at the given index.

__len__() → int

Return the number of the bin ranges.

__iter__() → Iterator[*Range*]

Iterate over the bin ranges.

bin_index(*ranges:* BinRanges, *value: float*) → int

Find the index of the bin range among `ranges` corresponding to `value`.

Requires

- `not math.isnan(value)`

Ensures

- `value < ranges[0].start result == -1`

  (Value not covered in ranges => bin not found)
- `value > ranges[-1].end result == -1`

  (Value not covered in ranges => bin not found)
- `ranges[0].start <= value <= ranges[-1].end 0 <= result < len(ranges)`

  (Value in the ranges => bin found)
- `result != -1 ranges[result].start <= value < ranges[result].end`

  (Index not found or it corresponds to the correct bin range)
  ```

class Histogram(*ranges:* BinRanges)

Represent a mutable histogram.

Establishes

- `all(count >= 0 for count in self.counts)`

Methods:

__init__(ranges)	Initialize the histogram with zero counts for `ranges`.
add(value)	Count the `value` in the corresponding bin.
items()	Iterate over (bin range, count of observations).

Attributes:

ranges	Bin ranges
counts	Count of observations for the given bin

__init__(*ranges:* BinRanges) → None

Initialize the histogram with zero counts for `ranges`.

Requires

- `len(ranges) > 0`

ranges

Bin ranges

counts

Count of observations for the given bin

add(*value: float*) → None

Count the `value` in the corresponding bin.

Requires

- `self.ranges[0].start <= value < self.ranges[-1].end`
- `not math.isnan(value)`

items() → Iterator[Tuple[*Range*, int]]

Iterate over (bin range, count of observations).

compute_histogram(*measurements: Sequence[*Measurement*]*) → List[Tuple[*Range*, int]]

Compute the histogram over `measurements`.

Returns

List of (bin range, count of observations for that bin)

Requires

- `len(measurements) > 0`

Ensures

- `len(measurements) == sum(item[1] for item in result)`

Problem 4

Analyze the booking data of a hotel.

The data is given as: * The room number, * The beginning of the booking interval (as an integer, the day of year), * Tne end of the booking interval (as an integer, the day of year), * The price of the room per day, * The price discount (as a floating point number between 0 and 100).

All bookings start and end in the same year.

The program answers the following questions: * Which room was booked the most frequently? (as number of bookings) * Which room was booked the longest (as number of days) * Which room brought in the most revenue? * What is the total revenue of the hotel for the year?

Classes:

Entry(room_number, start, end, ...)	Represent an entry in the booking data.

Functions:

most_booked_room(entries)	Find the number of the most booked room in the `entries`.
longest_booked_room(entries)	Find the room booked for the longest accumulated time according to `entries`.
room_with_most_revenue(entries)	Find the room which brought in the most revenue according to `entries`.
total_revenue(entries)	Compute the total revenue of the hotel based on `entries`.

class Entry(*room_number: int*, *start: int*, *end: int*, *price_per_day: float*, *price_discount: float*)

Represent an entry in the booking data.

Methods:

__init__(room_number, start, end, ...)	Initialize with the given values.
duration()	Compute the duration of the stay.
__repr__()	Return repr(self).

__init__(*room_number: int*, *start: int*, *end: int*, *price_per_day: float*, *price_discount: float*) → None

Initialize with the given values.

Requires

- `not math.isnan(price_per_day) and price_per_day < 1e300`

  (Reasonable bounds to avoid NaN's and inf's)

- `not math.isnan(price_discount) and price_discount < 1e300`

  (Reasonable bounds to avoid NaN's and inf's)

- `0 <= price_discount <= 100`

  (The price discount is given as a percentage.)

- `price_per_day > 0`

- `start <= end`

  (The start and end must occur in the same year)

- `1 <= end <= 366`

  (The end denotes the day of year.)

- `1 <= start <= 366`

  (The start denotes the day of year.)

- `room_number > 0`

duration() → int

Compute the duration of the stay.

Ensures

- `result > 0`

__repr__() → str

Return repr(self).

most_booked_room(*entries: List[*Entry*]*) → int

Find the number of the most booked room in the `entries`.

Requires

- `len(entries) > 0`

Ensures

- `result in {entry.room_number for entry in entries}`

longest_booked_room(*entries: Collection[*Entry*]*) → int

Find the room booked for the longest accumulated time according to `entries`.

Requires

- `len(entries) > 0`

Ensures

- `result in {entry.room_number for entry in entries}`

room_with_most_revenue(*entries: Collection[*Entry*]*) → int

Find the room which brought in the most revenue according to `entries`.

Requires

- `len(entries) > 0`

Ensures

- `result in {entry.room_number for entry in entries}`

total_revenue(*entries: Collection[*Entry*]*) → float

Compute the total revenue of the hotel based on `entries`.

Requires

- `len(entries) > 0`

Ensures

- `result >= 0`

2.3.6 Solutions to Exercise 6

These are the auto-documented solutions to the Exercise 6 used during the lecture "Introduction to Programming" at ETH Zurich (Switzerland) in Fall 2019.

The text of the exercise is available (in German) at: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2019/0027_Intro/Homework/u6.pdf

Problem 1

Parse the time in Swiss german.

The time `t` is given in `hh:mm` (`hh` for hours, `mm` for minutes). Translate it to Swiss german.

The "Swiss time" goes only from 1 to 12, but different periods of the day are indicated ("znacht", "am morge" *etc.*). Here's the pattern:

```
00:00 -> 12i znacht
01:45 -> viertel vor 2 znacht
09:25 -> 5 vor halbi 10i am morge
12:01 -> 1 ab 12i am mittag
16:46 -> 14 vor 5i am namittag
21:51 -> 9 vor 10i am abig
22:37 -> 7 ab halbi 11i znacht
```

The period of day is determined as follows:

- "Am morge" is after 6am and before 12pm.
- 12pm is "mittag".
- "Namittag" is till 6pm.
- "Abig" goes till 10pm.
- The remainder of the day is "znacht".

If the number of minutes (`mm`) is greater-equal 25, the hours are incremented by 1 ("5 vor halbi 10i").

If there are less than 25 minutes, you say "ab", but otherwise "vor". Between the minutes 25 and 39, you say "vor halbi" or "ab halbi". If it is exactly 15, 30 or 45, you say "viertel ab", "halbi" or "viertel vor", respectively.

For the hours after 3, you need to put a suffix "i" ("viertel vor 2", but "1 ab 12i").

Functions:

time_in_swiss_german(hour, minute)	Translate the given time of the day into Swiss german.

time_in_swiss_german(*hour: int*, *minute: int*) → str

Translate the given time of the day into Swiss german.

Requires

- `0 <= minute < 60`
- `0 <= hour < 23`

Problem 4

Implement a linked list which stores integers.

Provide the following operations:

- `add_first`,
- `remove_first`,
- `remove_last`,
- `clear`,
- `is_empty`,
- `get` (at index),
- `set` (at index), and
- `iterate` (over all the elements of the list).

(We substantially shortened the text of this problem to only give the crux of the problem and leave out the parts with user input/output. Please refer to the original exercise in German for exact details.)

Classes:

Node(value, next_node)	Represent a node of a linked list containing integers.
Cursor(linked_list)	Provide a cursor to iterate manually over a linked list.
LinkedList([values])	Provide a linked list.

class Node(*value: int*, *next_node: Optional[Node]*)

Represent a node of a linked list containing integers.

Methods:

__init__(value, next_node)	

__init__(*value: int*, *next_node: Optional[Node]*) → None

class Cursor(*linked_list:* LinkedList)

Provide a cursor to iterate manually over a linked list.

Methods:

__init__(linked_list)	Initialize the cursor to point to the beginning of the `linked_list`.
value()	Retrieve the value point to by the cursor.
set_value(value)	Set the value in the linked list at the cursor.
move()	Move the cursor to the next element.
done()	Return `True` if the cursor is past the end of linked list.
is_last()	Return `True` if the cursor points to the last element of the list.
remove()	Remove the node (and the value, respectively) at the cursor.

`__init__`(*linked_list:* LinkedList) → None

Initialize the cursor to point to the beginning of the `linked_list`.

`value`() → int

Retrieve the value point to by the cursor.

Requires

- `not self.done()`

`set_value`(*value: int*) → None

Set the value in the linked list at the cursor.

Requires

- `not self.done()`

`move`() → None

Move the cursor to the next element.

Requires

- `not self.done()`

`done`() → bool

Return `True` if the cursor is past the end of linked list.

`is_last`() → bool

Return `True` if the cursor points to the last element of the list.

`remove`() → int

Remove the node (and the value, respectively) at the cursor.

Requires

- `not self.done()`

OLD

- `.values_without = list(_values_except_node(self._linked_list, self._node))`
- `.count = self._linked_list.count()`

Ensures

- `list(self._linked_list.values()) == OLD.values_without`

`class LinkedList`(*values: Optional[Iterable[int]] = None*)

Provide a linked list.

Establishes

- `len(list(self.values())) == self.count()`
- `self._last is not None self._last.next_node is None`
- `len(list(self.values())) != 0 ^ self.is_empty()`
- `self.is_empty() ^ (self._first is not None and self._last is not None)`

Methods:

__init__([values])	Initialize the list by populating it with the given `values`.
add_first(value)	Prepend the `value` to the list.
add_last(value)	Append the `value` to the list.
count()	
remove_first()	Remove first element of the list.
remove_last()	Remove the last element of the list.
clear()	Remove all elements in the list.
is_empty()	
get(index)	Retrieve the `index`-th element of the list.
set(index, value)	Set the `index`-th element to `value`.
values()	Iterate over all the values in the list.
cursor()	Get a cursor pointing to the beginning of the list.

__init__(*values: Optional[Iterable[int]] = None*) → None

Initialize the list by populating it with the given `values`.

add_first(*value: int*) → None

Prepend the `value` to the list.

OLD

- `.values` = `list(self.values())`
- `.count` = `self.count()`

Ensures

- `self.count() == OLD.count + 1`
- `not self.is_empty()`
- `[value] + OLD.values == list(self.values())`

add_last(*value: int*) → None

Append the `value` to the list.

OLD

- `.values` = `list(self.values())`
- `.count` = `self.count()`

Ensures

- `self.count() == OLD.count + 1`
- `not self.is_empty()`
- `OLD.values + [value] == list(self.values())`

count() → int

remove_first() → int

Remove first element of the list.

Requires

- `not self.is_empty()`

OLD

- `.count` = `self.count()`
- `.values` = `list(self.values())`

Ensures

- `OLD.values[0] == result`
- `self.count() == OLD.count - 1`
- `OLD.values[1:]  == list(self.values())`

remove_last() → int

Remove the last element of the list.

Requires

- `not self.is_empty()`

OLD

- `.values` = `list(self.values())`
- `.count` = `self.count()`

Ensures

- `OLD.values[-1] == result`
- `self.count() == OLD.count - 1`
- `OLD.values[:-1] == list(self.values())`

clear() → None

Remove all elements in the list.

Requires

- `not self.is_empty()`

Ensures

- `self.count() == 0`
- `self.is_empty()`

is_empty() → bool

get(*index: int*) → int

Retrieve the `index`-th element of the list.

Requires

- `0 <= index < self.count()`

Ensures

- `list(self.values())[index] == result`

set(*index: int*, *value: int*) → None

Set the `index`-th element to `value`.

Requires

- `0 <= index < self.count()`

Ensures

- `list(self.values())[index] == value`

values() → Iterator[int]

Iterate over all the values in the list.

cursor() → *Cursor*

Get a cursor pointing to the beginning of the list.

Problem 5

Compute the angles of the clock hands for a given time of the day.

The 00:00:00 corresponds to the angle zero, increasing to the right. (This was *not* part of the original problem statement, but we introduce it here for simplicity of the solution.)

(We change slightly the statement so that it focuses on what we thought the "core" part of the problem. The original problem additionally concerns the drawing of the clock hands in the GUI which we intentionally leave out as Java-specific and not relevant in terms of contracts.)

Functions:

compute_angles(hour, minute, second)	Compute the angles of the clock hands for a given time of the day.

compute_angles(*hour: int*, *minute: int*, *second: int*) → Tuple[float, float, float]

Compute the angles of the clock hands for a given time of the day.

Requires

- `0 <= second < 60`
- `0 <= minute < 60`
- `0 <= hour <= 23`

Ensures

- ```
  not (minute == 0 and second == 0)
  or (result[1] == 0.0 and result[2] == 0.0)
  ```

- `second / 60 * 360 <= result[2] < (second + 1) / 60 * 360`

  (Second hand between two second ticks)

- `minute / 60 * 360 <= result[1] < (minute + 1) / 60 * 360`

  (Minute hand between two minute ticks)

- ```
  (
          clock_hour := hour if hour < 12 else hour - 12,
          clock_hour / 12 * 360 <= result[0] < (clock_hour + 1) / 12 *␣
  ↪360
  )[1]
  ```

  (Hour hand between two hour ticks)

- ```
  not (hour == 0 and minute == 0 and second > 0)
  or result[0] > 0 and result[1] > 0 and result[2] > 0
  ```

  (All hands of a clock move when the hand for seconds moves)

- ```
  not (hour == 0 and minute == 0 and second == 0)
  or result[0] == 0 and result[1] == 0 and result[2] == 0
  ```

  (Angles start from 00:00:00)

- `all(0 <= angle < 360 for angle in result)`

2.3.7 Solutions to Exercise 7

These are the auto-documented solutions to the Exercise 7 used during the lecture "Introduction to Programming" at ETH Zurich (Switzerland) in Fall 2019.

The text of the exercise is available (in German) at: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2019/0027_Intro/Homework/u7.pdf

Problem 2

Add an operation `split` to the linked list from Exercise 6, Problem 4.

The operation `split` takes in an argument `n` and removes all the items from the list which are greater-equal `n`.

The operation creates an additional list and inserts in it all the removed elements (in the same order as in the original list). The resulting second list is finally returned to the caller.

(The original problem statements includes an additional requirement that the caller should be able to iterate over the original list by adding extra pointers `old_next` to the original nodes. We deliberately remove this part of the problem as it introduces, in our opinion, complexity in the code with no or only marginal insights about the nature of the code contracts).

Functions:

same_order(lst, old_values)	Check that the values in `lst` follow the order of the `old_values`.
split(lst, n)	Remove the elements greater-equal `n` from the `lst`.

same_order(*lst: List[int]*, *old_values: List[int]*) → bool

Check that the values in `lst` follow the order of the `old_values`.

Requires

- `len(lst) <= len(old_values)`

split(*lst:* LinkedList, *n: int*) → *LinkedList*

Remove the elements greater-equal `n` from the `lst`.

return

The elements removed from the `lst`

OLD

- `.values` = `list(lst.values())`

- `.count = lst.count()`

Ensures

- `lst.count() + result.count() == OLD.count`
- `all(value >= n for value in result.values())`
- `all(value < n for value in lst.values())`
- `same_order(list(lst.values()), OLD.values)`
- `same_order(list(result.values()), OLD.values)`

Problem 4

Write a program to draw a tree recursively.

Each branch forks in two, one sub-branch that goes clockwise and the other that goes counter-clockwise. The tree is drawn in multiple steps. Each step is defined by a tuple `(x, y, alpha, l)`:

- Starting point of the actual segment `(x, y)`,
- The direction of the segment `alpha`,
- The length of the segment `l`.

The recursion stops when `l < 10` and ends with a leaf.

The forking of a branch occurs as follows. The sub-branch that goes clockwise follows with `l' = 0.8 * l` and `alpha' = alpha + PI / 5`. The sub-branch that goes counter-clockwise follows with `l'' = 0.6 * l` and `alpha'' = alpha - PI / 3`.

The starting point of the sub-branches is the end point of the preceding segment.

Start the program with `x = SIZE / 2, y = SIZE`, `l_0 = 100` and `alpha_0 = PI / 2`, where `SIZE` is the size of the drawing canvas.

2.3.8 Solutions to Exercise 8

These are the auto-documented solutions to the Exercise 8 used during the lecture "Introduction to Programming" at ETH Zurich (Switzerland) in Fall 2019.

The text of the exercise is available (in German) at: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2019/0027_Intro/Homework/u8.pdf

Problem 1

Consider two strings, `s` and `t` and a non-negative integer `k`.

Check if the characters in `t` occur in `s` in an arbitrary order as a subsequence. The distance between two matching characters from `t` in `s` must not be larger than `k`.

The distance between two characters is given as the number of characters in-between. For example, the distance between the characters "a" and "b" in "a12345b" is 5.

Here are a couple of examples:

- `s = "abbbc", t = "cab", k = 1`. The result is `True`.
- `s = "abbbbc", t = "cab", k = 1`. The result if `False`. The distance between either "a" and "b" or "b" and "c" is larger than 1.

- `s = "abc"`

Functions:

matches(src, chars, dist)	**requires**
stretch(src, chars, initialdist, dist)	

matches(*src: str*, *chars: str*, *dist: int*) → bool

Requires

- `dist >= 0`

Ensures

- `chars in src result is True`
- ```
  (result is True)
  if (sorted(chars) in sorted(src))
  else True
  ```
- `len(chars) > len(src) result is False`

stretch(*src: str*, *chars: Counter[str]*, *initialdist: int*, *dist: int*) → bool

Problem 2

List all the sub-sequences of length `n` contained in a string `s`.

You obtain a sub-sequence by "erasing" the characters in `s`.

For example, here are all the sub-sequences with `n = 2` of `"apple"`:

```
"ap"
"al"
"ae"
"pp"
"pl"
"pe"
"le"
```

Functions:

is_subsequence(subtext, text)	Check if the `subtext` is a subsequence of `text`.
list_subsequences(text, length)	List all subsequences of size `length` in `text`.

is_subsequence(*subtext: str*, *text: str*) → bool

Check if the `subtext` is a subsequence of `text`.

```
>>> is_subsequence('pe', 'apple')
True
```
```

```
>>> is_subsequence('ep', 'apple')
False
```

Requires

- `len(subtext) <= len(text)`

`list_subsequences`(*text: str*, *length: int*) → Set[str]

List all subsequences of size `length` in `text`.

```
>>> sorted(list_subsequences("apple", 2))
['ae', 'al', 'ap', 'le', 'pe', 'pl', 'pp']
```

Requires

- `0 <= length <= len(text)`

Ensures

- ```
  not (
      length > 0 and len(text) > 0 and len(set(text)) == len(text)
  )
  or len(result) == math.comb(len(text), length)
  ```

  (If all the characters are unique, the number of substrings equals the count of binary sequences `len(text)` long with `length` bits set)

- `all(is_subsequence(item, text) for item in result)`
- `all(len(item) == length for item in result)`

Problem 3

Simulate a shopping queue in discrete steps.

Here are the inputs:

- Number of checkouts with the respective efficiency `p_0, p_1, ..., p_n` with `0 <= p_i < 1`. In every step, the cashier `i` scans with a probability `p_i` an item from the shopping cart of the customer at the front of the queue. When all the items from the shopping cart have been scanned, the customer leaves the queue.
- A new customer shows up with a probability `e, 0 <= e < 1` at every step and enters one of the queues at random.
- The size of the shopping cart `w`. Every customer has a random number of items between 0 and `w`.

(We removed the feature "laziness factor" from the original problem statement. The feature was not very clear to us, and would not have had much impact on the contracts.)

Provide the following metrics of the simulation:

- `finished`: Number of people who left the queue at the end of the simulation.
- `avg_queue_lengths`: A list of the average length of each queue during the simulation
- `max_queue_lengths`: A list of the maximum length of each queue during the simulation
  ```

Classes:

Probability(value)	Represent a probability value.
Specs(checkout_efficiencies, ...)	Specify the parameters of the simulation.
Stats(finished, avg_queue_lengths, ...)	Structure the results of the simulation.
Customer(items_in_cart)	Represent the current state of the customer in the simulation.

Functions:

simulate(specs, steps)	Simulate in `steps` number of iterations according to the `specs`.

class Probability(*value: float*)

Represent a probability value.

Methods:

__new__(cls, value)	Enforce the properties of a probability on `value`.

static __new__(*cls*, *value: float*) → *Probability*

Enforce the properties of a probability on `value`.

Requires

- `0 <= value < 1`

class Specs(*checkout_efficiencies: List[*Probability*]*, *new_customer_probability:* Probability, *max_cart_size: int*)

Specify the parameters of the simulation.

Methods:

__init__(checkout_efficiencies, ...)	Initialize with the given values.
__repr__()	Represent the specs for easier debugging.

__init__(*checkout_efficiencies: List[*Probability*]*, *new_customer_probability:* Probability, *max_cart_size: int*) → None

Initialize with the given values.

Requires

- `len(checkout_efficiencies) > 0`
- `max_cart_size >= 1`

__repr__() → str

Represent the specs for easier debugging.

class Stats(*finished: int*, *avg_queue_lengths: List[float]*, *max_queue_lengths: List[float]*)

Structure the results of the simulation.

Methods:

__init__(finished, avg_queue_lengths, ...)	Initialize with the given values.

__init__(*finished: int*, *avg_queue_lengths: List[float]*, *max_queue_lengths: List[float]*) → None

Initialize with the given values.

Requires

- ```
  all(
      a_max >= 0
      for a_max in max_queue_lengths
  )
  ```
- ```
  all(
      an_avg >= 0
      for an_avg in avg_queue_lengths
  )
  ```
- `finished >= 0`

class Customer(*items_in_cart: int*)

Represent the current state of the customer in the simulation.

Methods:

__init__(items_in_cart)	Initialize with the given values.

__init__(*items_in_cart: int*) → None

Initialize with the given values.

Requires

- `items_in_cart >= 0`

simulate(*specs:* Specs, *steps: int*) → *Stats*

Simulate in `steps` number of iterations according to the `specs`.

Requires

- `steps >= 1`

Ensures

- `len(result.max_queue_lengths) == len(specs.checkout_efficiencies)`
- `len(result.avg_queue_lengths) == len(specs.checkout_efficiencies)`
- ```
  all(
      avg_queue_length <= max_queue_length
      for avg_queue_length, max_queue_length in zip(
          result.avg_queue_lengths, result.max_queue_lengths)
  )
  ```
  ```

Problem 5

Simulate the wolf fleeing a city.

A city is given as a grid of `n x n` crossroads. At each crossroad, the wolf picks a random direction. The grid size `n` is greater than 1 and odd.

The wolf starts at `(0, 0)`. The wolf escaped the city when he reaches the border of the grid.

The wolf can never visit the same crossroad twice. If there are no more options, the wolf is shot by the hunters and dies.

Estimate the probability of the wolf escaping the city.

(We deliberately exclude the parts of the exercise concerning the drawing of the paths in the GUI.)

Classes:

Position(x, y)	Represent the current position of the wolf on the grid.

Functions:

list_next_positions(pos)	List all the possible next positions based on the current position `pos`.
simulate(trials, grid_size)	Simulate `trials` number of wolf journeys on the quadratic ``grid_size`` grid.

class Position(*x: int*, *y: int*)

Represent the current position of the wolf on the grid.

Methods:

__init__(x, y)	Initialize with the given values.
__repr__()	Represent the instance as a string for debugging.

Attributes:

x	X-coordinate of the cell
y	Y-coordinate of the cell

__init__(*x: int*, *y: int*) → None

Initialize with the given values.

x: Final[int]

X-coordinate of the cell

y: Final[int]

Y-coordinate of the cell

__repr__() → str

Represent the instance as a string for debugging.

list_next_positions(*pos:* Position) → Sequence[*Position*]

List all the possible next positions based on the current position `pos`.

Ensures

- ```
  all(
      (next_pos.x == pos.x and next_pos.y != pos.y)
      ^ (next_pos.x != pos.x and next_pos.y == pos.y)
      for next_pos in result
  )
  ```

  (Next is either in x- or in y-direction)

- ```
  all(
      not (next_pos.y != pos.y) or (abs(next_pos.y - pos.y) == 1)
      for next_pos in result
  )
  ```

  (Next is at most 1 field in y-direction)

- ```
  all(
      not (next_pos.x != pos.x) or (abs(next_pos.x - pos.x) == 1)
      for next_pos in result
  )
  ```

  (Next is at most 1 field in x-direction)

- `pos not in result`
- `len(result) == 4`

simulate(*trials: int*, *grid_size: int*) → float

Simulate `trials` number of wolf journeys on the quadratic ``grid_size`` grid.

Requires

- `grid_size % 2 == 1`
- `grid_size > 1`
- `trials > 0`

Ensures

- `0 <= result <= 1`

2.3.9 Solutions to Exercise 9

These are the auto-documented solutions to the Exercise 9 used during the lecture "Introduction to Programming" at ETH Zurich (Switzerland) in Fall 2019.

The text of the exercise is available (in German) at: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2019/0027_Intro/Homework/u9.pdf
  ```

Problem 2

Reverse a linked list (extension from Exercise 6, Problem 4).

Classes:

ReversibleLinkedList([values])	Extend a `LinkedList` with reverse operation.

class ReversibleLinkedList(*values: Optional[Iterable[int]] = None*)

Extend a `LinkedList` with reverse operation.

Establishes

- `len(list(self.values())) == self.count()`
- `self._last is not None self._last.next_node is None`
- `len(list(self.values())) != 0 ^ self.is_empty()`
- `self.is_empty() ^ (self._first is not None and self._last is not None)`

Methods:

reverse()	Reverse the elements of the list.
__eq__(value, /)	Return self==value.
__ge__(value, /)	Return self>=value.
__gt__(value, /)	Return self>value.
__init__([values])	Initialize the list by populating it with the given `values`.
__le__(value, /)	Return self<=value.
__lt__(value, /)	Return self<value.
__ne__(value, /)	Return self!=value.
__str__()	Return str(self).
add_first(value)	Prepend the `value` to the list.
add_last(value)	Append the `value` to the list.
clear()	Remove all elements in the list.
count()	
cursor()	Get a cursor pointing to the beginning of the list.
get(index)	Retrieve the `index`-th element of the list.
is_empty()	
remove_first()	Remove first element of the list.
remove_last()	Remove the last element of the list.
set(index, value)	Set the `index`-th element to `value`.
values()	Iterate over all the values in the list.

Attributes:

__invariants__

reverse() → None

Reverse the elements of the list.

OLD

- .count = self.count()
- .values = list(self.values())

Ensures

- self.count() == OLD.count
- list(self.values()) == list(reversed(OLD.values))

__eq__(*value*, /)

Return self==value.

__ge__(*value*, /)

Return self>=value.

__gt__(*value*, /)

Return self>value.

__init__(*values: Optional[Iterable[int]] = None*) → None

Initialize the list by populating it with the given values.

__invariants__ = [<icontract._types.Contract object>, <icontract._types.Contract object>, <icontract._types.Contract object>, <icontract._types.Contract object>]

__le__(*value*, /)

Return self<=value.

__lt__(*value*, /)

Return self<value.

__ne__(*value*, /)

Return self!=value.

__str__()

Return str(self).

add_first(*value: int*) → None

Prepend the value to the list.

OLD

- .values = list(self.values())
- .count = self.count()

Ensures

- self.count() == OLD.count + 1
- not self.is_empty()
- [value] + OLD.values == list(self.values())

add_last(*value: int*) → None

Append the value to the list.

OLD

- `.values = list(self.values())`
- `.count = self.count()`

Ensures

- `self.count() == OLD.count + 1`
- `not self.is_empty()`
- `OLD.values + [value] == list(self.values())`

clear() → None

Remove all elements in the list.

Requires

- `not self.is_empty()`

Ensures

- `self.count() == 0`
- `self.is_empty()`

count() → int

cursor() → *Cursor*

Get a cursor pointing to the beginning of the list.

get(*index: int*) → int

Retrieve the `index`-th element of the list.

Requires

- `0 <= index < self.count()`

Ensures

- `list(self.values())[index] == result`

is_empty() → bool

remove_first() → int

Remove first element of the list.

Requires

- `not self.is_empty()`

OLD

- `.count = self.count()`
- `.values = list(self.values())`

Ensures

- `OLD.values[0] == result`
- `self.count() == OLD.count - 1`
- `OLD.values[1:]  == list(self.values())`

remove_last() → int

Remove the last element of the list.

Requires

- `not self.is_empty()`

OLD

- `.values` = `list(self.values())`
- `.count` = `self.count()`

Ensures

- `OLD.values[-1] == result`
- `self.count() == OLD.count - 1`
- `OLD.values[:-1] == list(self.values())`

set(*index: int*, *value: int*) → None

Set the `index`-th element to `value`.

Requires

- `0 <= index < self.count()`

Ensures

- `list(self.values())[index] == value`

values() → Iterator[int]

Iterate over all the values in the list.

Problem 3

Parse the file containing the list of bonus flying miles.

Here is an example:

```
Michaela Meier
LX326 05.12.2016 ECONOMY
LX317 10.01.2017 ECONOMY
A3851 12.05.2017 BUSINESS
LX8 12.10.2017 FIRST 4433
.
Stefan Oliver Schmid
LX4150 19.10.2017 BUSINESS 6404
.
```

The list consists of blocks. Each block begins with the name of the flier. The name is then followed by the list of flights. For each flight, the flight number, the flight date (in `dd.mm.yyyy`), the class of the flight ("ECONOMY", "BUSINESS", "FIRST") are given. If it is an inter-continental flight, the number of miles follows the class of flight.

If no miles are given, you can assume a "flat-rate" of 125 miles. The miles flown in "BUSINESS" are counted double and the miles in "FIRST" triple, respectively.

Output the number of miles per person.

For the aforementioned example:

```
Michaela Meier: 13799
Stefan Oliver Schmid: 12808
```

Classes:

ClassOfFlight(value)	Enumerate the different classes of flights.
Flight(number, date, class_of_flight, miles)	Represent an entry in the flight data.
Block(name, flights)	Represent a block of flight entries tied to the flier.

Data:

STR_TO_CLASS_OF_FLIGHT	Map literal value enumeration.
FLIGHT_RE	Express the flight entry in the text data.

Functions:

compile_flight_re()	Compile the regular expression that expresses the flight entry in the data.
parse_block(lines)	Parse the given block of the input data given as text `lines`.
parse(lines)	Parse the input data given as text `lines` into structured blocks.
compute_totals(blocks)	Compute the total miles collected by the flier.

class ClassOfFlight(*value*)

Enumerate the different classes of flights.

Attributes:

ECONOMY
BUSINESS
FIRST

ECONOMY = 'ECONOMY'

BUSINESS = 'BUSINESS'

FIRST = 'FIRST'

STR_TO_CLASS_OF_FLIGHT: Mapping[str, *ClassOfFlight*] = {'BUSINESS': ClassOfFlight.BUSINESS, 'ECONOMY': ClassOfFlight.ECONOMY, 'FIRST': ClassOfFlight.FIRST}

Map literal value enumeration.

class Flight(*number: str*, *date: date*, *class_of_flight:* ClassOfFlight, *miles: Optional[int]*)

Represent an entry in the flight data.

Methods:

__init__(number, date, class_of_flight, miles)	Initialize with the given values.

__init__(*number: str*, *date: date*, *class_of_flight:* ClassOfFlight, *miles: Optional[int]*) → None

Initialize with the given values.

Requires

- `miles is not None miles > 0`

Ensures

- `miles is None self.miles == 125`

(If `miles` is not given, set to the default value.)

class Block(*name: str*, *flights: Sequence[*Flight*]*)

Represent a block of flight entries tied to the flier.

Methods:

__init__(name, flights)	Initialize with the given values.

__init__(*name: str*, *flights: Sequence[*Flight*]*)

Initialize with the given values.

compile_flight_re() → Pattern

Compile the regular expression that expresses the flight entry in the data.

FLIGHT_RE = re.compile('^(?P<number>[a-zA-Z0-9]+) (?P<date>[0-9]{2}.[0-9]{2}.[0-9]{4}) (?P<class_of_flight>ECONOMY|BUSINESS|FIRST)((?P<miles>[0-9]+))?$')

Express the flight entry in the text data.

parse_block(*lines:* Lines) → *Block*

Parse the given block of the input data given as text `lines`.

Requires

- `all(FLIGHT_RE.match(line) for line in lines[1:])`
- `len(lines) >= 1`

Ensures

- `len(lines) == 1 len(result.flights) == 0`
- `result.name == lines[0]`

parse(*lines:* Lines) → List[*Block*]

Parse the input data given as text `lines` into structured blocks.

Ensures

- `len(lines) > 0 len(result) > 0`
- `len(lines) == 0 len(result) == 0`

compute_totals(*blocks: List[*Block*]*) → MutableMapping[str, int]

Compute the total miles collected by the flier.

Returns

Flier name miles collected

Ensures

- `len(blocks) != 0 len(result) > 0`

- `set(block.name for block in blocks) == set(result.keys())`

  (All people considered)

- `all(value >= 0 for value in result.values())`

Problem 4

Implement an "AI" for a game of word guessing.

(We excluded Exercise 5, Problem 2 as too simple for the contracts. However, the problem is re-introduced in this problem as well, so we merge the two problems in one.

Additionally, we exclude the parts a) and b) from the original problem as they are not really interesting for the contracts but are rather focused on teaching concepts of inheritance. We therefore focus on part c) of the problem, which is much more complex and algorithmically more involving.)

The computer picks randomly a word `w` from a pre-defined list of words. In each round of the game, the player needs to guess a word. Based on the guess `g`, the computer returns one or more possible hints:

- `w` starts with `g`.
- `w` ends with `g`.
- `w` contains `g`.
- `w` does not contain `g`.

The game ends when the player hits the word.

The "AI" guesses the word automatically (given the pre-defined list of words). How many rounds on average does it take the artificial player to finish the game?

2.3.10 Solutions to Exercise 10

We do not provide any auto-documented solutions for this exercises as we decided to skip all of the corresponding problems. The problems were judged for exclusion on problem-by-problem basis. Please see *Details about the exclusion of problems in Exercise 10* why each individual problem has been excluded.

The text of the exercise is available (in German) at: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2019/0027_Intro/Homework/u10.pdf

2.3.11 Solutions to Exercise 11

These are the auto-documented solutions to the Exercise 11 used during the lecture "Introduction to Programming" at ETH Zurich (Switzerland) in Fall 2019.

The text of the exercise is available (in German) at: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2019/0027_Intro/Homework/u11.pdf

Problem 1

Analyze the grades of the students.

Here's an example of the data

```
111111004  5.0  5.0  6.0
111111005  3.75 3.0  4.0
111111006  4.5  2.25 4.0
```

Every line represents a grading of a student. It starts with her matriculation number, followed by space-delimited grades (between 1.0 and 6.0, floats). The grades correspond to lectures 1, 2 and 3, respectively.

Provide the function `critical` which accepts two arguments, `bound1` and `bound2`. The function lists all the students which have "critical" grades. A student should appear only once in the list.

A student is "critical" if the grade for the first lecture is smaller-equal `bound1` and the sum of the grades for the lecture 2 and 3 is smaller than `bound2`.

On the above example, `critical(4, 8)` gives:

```
111111005
```

Provide the function `top` which lists the students with the best grades. The parameter `limit` determines the number of the "top" students. If the number of students is less than `limit`, return the list of all the students.

A student should appear only once in the resulting list. The students are compared based on the sum of the grades in all the three lectures. If the sum of grades is equal for two students, the order in the list is undefined (*i.e.* does not matter).

On the above example, `top(2)` might return both the output:

```
111111004
111111005
```

and:

```
111111004
111111006
```

(Both outputs are valid.)

The parameter `limit` is always greater than 0. Both `bound1` and `bound2` are expected in the range `[0.0, 100.0]`.

Data:

ALL_GRADES	List all possible grades.
ALL_GRADES_SET	Provide a set of all possible grades.
GRADING_RE	Express a grading entry for a student as a text.

Classes:

Grade(value)	Represent a grade in Swiss educational system.
Grading(identifier, grade1, grade2, grade3)	Represent the grading of a student.

Functions:

parse(lines)	Parse the grading entries given as `lines`.
critical(gradings, bound1, bound2)	List critical gradings among the `gradings` based on bound1 and bound2.
top(gradings, limit)	Find the top `limit` students among the `gradings`.

ALL_GRADES = [Decimal('1.00'), Decimal('1.25'), Decimal('1.50'), Decimal('1.75'), Decimal('2.00'), Decimal('2.25'), Decimal('2.50'), Decimal('2.75'), Decimal('3.00'), Decimal('3.25'), Decimal('3.50'), Decimal('3.75'), Decimal('4.00'), Decimal('4.25'), Decimal('4.50'), Decimal('4.75'), Decimal('5.00'), Decimal('5.25'), Decimal('5.50'), Decimal('5.75'), Decimal('6.00')]

List all possible grades.

ALL_GRADES_SET = {Decimal('1.00'), Decimal('1.25'), Decimal('1.50'), Decimal('1.75'), Decimal('2.00'), Decimal('2.25'), Decimal('2.50'), Decimal('2.75'), Decimal('3.00'), Decimal('3.25'), Decimal('3.50'), Decimal('3.75'), Decimal('4.00'), Decimal('4.25'), Decimal('4.50'), Decimal('4.75'), Decimal('5.00'), Decimal('5.25'), Decimal('5.50'), Decimal('5.75'), Decimal('6.00')}

Provide a set of all possible grades.

class Grade(*value: Decimal*)

Represent a grade in Swiss educational system.

Methods:

__new__(cls, value)	Enforce the grade properties on `value`.
__le__(other)	Return `True` if `self` is smaller than `other`.
__add__(other)	Add `self` and `other`.

static __new__(*cls*, *value: Decimal*) → *Grade*

Enforce the grade properties on `value`.

Requires

- `not value.is_nan()`
- `value in ALL_GRADES_SET`

__le__(*other: Union[Decimal, float, Rational,* Grade*]*) → bool

Return `True` if `self` is smaller than `other`.

__add__(*other: Union[Decimal, int,* Grade*]*) → Decimal

Add `self` and `other`.

class Grading(*identifier: str*, *grade1:* Grade, *grade2:* Grade, *grade3:* Grade)

Represent the grading of a student.

Methods:

__init__(identifier, grade1, grade2, grade3)	Initialize with the given values.
sum_grades()	Sum all grades of the student.

__init__(*identifier: str*, *grade1:* Grade, *grade2:* Grade, *grade3:* Grade) → None

Initialize with the given values.

sum_grades() → Decimal

Sum all grades of the student.

Ensures

- `3 * MIN_GRADE <= result <= 3 * MAX_GRADE`

GRADING_RE = re.compile('([a-zA-Z0-9]+) +(1.0|1.25|1.5|1.75|2.0|2.25|2.5|2.75|3.0|3.25|3.5|3.75|4.0|4.25|4.5|4.75|5.0|5.25|5.5|5.75|6.0) +(1.0|1.25|1.5|1.75|2.0|2.25|2.5|2.75|3.0|3.25|3.5|3.75|4.0|4.25|4.5|4.75|5.0|5.25|5.5|)

Express a grading entry for a student as a text.

Note: The function `re.Pattern.__repr__` truncates at 200 characters so that the pattern in the docs (based on `__repr__` function) is possibly incorrect. See Python issue #13592.

---

parse(*lines:* Lines) → List[*Grading*]

Parse the grading entries given as `lines`.

Requires

- `all(GRADING_RE.fullmatch(line) for line in lines)`

Ensures

- `len(result) == len(lines)`
- ```
  (
      identifiers := [grading.identifier for grading in result],
      len(identifiers) == len(set(identifiers)),
  )[1]
  ```

  (Unique identifiers)

critical(*gradings: List[*Grading*]*, *bound1:* Grade, *bound2: Decimal*) → List[*Grading*]

List critical gradings among the `gradings` based on `bound1` and `bound2`.

Note that `bound1` and `bound2` have special semantics. Please consult the text of the problem.

Requires

- ```
  (
          identifiers := [grading.identifier for grading in gradings],
          len(identifiers) == len(set(identifiers))
  )[1]
  ```

  (Students appear only once)

Ensures

- ```
  (
          identifiers := [grading.identifier for grading in result],
          len(identifiers) == len(set(identifiers))
  )[1]
  ```

  (Students appear only once)
  ```

`top`(*gradings: List[*Grading*]*, *limit: int*) → List[*Grading*]

Find the top `limit` students among the `gradings`.

Requires

- `limit > 0`
- ```
(
        identifiers := [grading.identifier for grading in gradings],
        len(identifiers) == len(set(identifiers))
)[1]
```

  (Students appear only once)

**Ensures**

- ```
all(
    result[i].sum_grades() >= result[i + 1].sum_grades()
    for i in range(len(result) - 1)
)
```
- `len(result) == min(limit, len(gradings))`
- ```
(
        identifiers := [grading.identifier for grading in result],
        len(identifiers) == len(set(identifiers))
)[1]
```

  (Students appear only once)

### Problem 2

Evaluate mathematical expressions.

Please see page 3 and page 4 of Exercise 11.

First, provide a tokenizer and a parser for the mathematical expressions.

Second, evaluate the mathematical expressions. The functions `cos`, `sin` and `tan` need to be supported. The evaluation function is given a dictionary of parameter values.

An exception should be raised if a parameter has not been specified.

**Classes:**
```

TokenKind(value)	Define the token kind.
TokenizationRule(kind, pattern)	Define a regular expression which specifies a token.
Token(value, start, end, kind)	Represent a token of the source code.
UnOp(value)	Represent unary operators.
Associativity(value)	Represent the associativity of a binary operator.
BinOpInfo(precedence, associativity)	Specify precedence and associativity.
BinOp(value)	Represent binary operators.
Identifier(value)	Represent an identifier of a variable or of a function.
Expr()	Represent a valid expression as an abstract syntax tree (AST).
Constant(value)	Represent a constant in the AST.
Variable(identifier)	Represent a variable in the AST.
UnaryOperation(target, operator)	Represent an unary operation in the AST.
BinaryOperation(left, operator, right)	Represent a binary operation in the AST.
Call(name, argument)	Represent a function call in the AST.
TokensWoWhitespace(tokens)	Represent tokens without whitespace.

Data:

TOKENIZATION	Define rules so that we can map token kind regular expression.
TOKENIZATION_MAP	Map token kind rule to be matched for that token kind.
IDENTIFIER_RE	Express an identifier of a variable or a function.

Functions:

tokenize(text)	Tokenize the given `text`.
tokens_to_text(tokens)	Serialize the `tokens` back into the original text.
parse_tokens(tokens)	Parse the given tokens into an expression.
unparse(expr)	Convert the AST given as `expr` back to the source code as text.
evaluate(expr, lookup)	Evaluate the given expression `expr` substituting variables with `lookup`.
collect_variables(expr)	Go recursively over the expression and collect the variable names.

class TokenKind(*value*)

Define the token kind.

Attributes:

NUM	Number literal
VAR	Variable (or function) identifier
OP	Operator
OPEN	Opening parenthesis
CLOSE	Closing parenthesis
WHITESPACE	Whitespace (including tabs *etc.*)

NUM = 1

Number literal

VAR = 2

Variable (or function) identifier

OP = 4

Operator

OPEN = 5

Opening parenthesis

CLOSE = 6

Closing parenthesis

WHITESPACE = 7

Whitespace (including tabs *etc.*)

class TokenizationRule(*kind:* TokenKind, *pattern: Pattern[str]*)

Define a regular expression which specifies a token.

Methods:

__init__(kind, pattern)	Initialize with the given values.
__repr__()	Represent the instance as a string for debugging.

__init__(*kind:* TokenKind, *pattern: Pattern[str]*) → None

Initialize with the given values.

__repr__() → str

Represent the instance as a string for debugging.

TOKENIZATION = [TokenizationRule(1,
re.compile('(inf|0|[1-9][0-9]*)(\\.[0-9]+)?(e[+\\-]?[0-9]+)?')), TokenizationRule(2,
re.compile('[a-zA-Z_][a-zA-Z_0-9]*')), TokenizationRule(4, re.compile('[+\\-*/^]')),
TokenizationRule(5, re.compile('\\(')), TokenizationRule(6, re.compile('\\)')),
TokenizationRule(7, re.compile('\\s+'))]

Define rules so that we can map token kind regular expression.

TOKENIZATION_MAP: Mapping[*TokenKind*, *TokenizationRule*] = {<TokenKind.NUM: 1>:
TokenizationRule(1, re.compile('(inf|0|[1-9][0-9]*)(\\.[0-9]+)?(e[+\\-]?[0-9]+)?')),
<TokenKind.VAR: 2>: TokenizationRule(2, re.compile('[a-zA-Z_][a-zA-Z_0-9]*')),
<TokenKind.OP: 4>: TokenizationRule(4, re.compile('[+\\-*/^]')), <TokenKind.OPEN: 5>:
TokenizationRule(5, re.compile('\\(')), <TokenKind.CLOSE: 6>: TokenizationRule(6,
re.compile('\\)')), <TokenKind.WHITESPACE: 7>: TokenizationRule(7, re.compile('\\s+'))}

Map token kind rule to be matched for that token kind.

class Token(*value: str*, *start: int*, *end: int*, *kind:* TokenKind)

Represent a token of the source code.

Methods:

__init__(value, start, end, kind)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on all the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*value: str*, *start: int*, *end: int*, *kind:* TokenKind) → None

Initialize with the given values.

Requires

- `start < end`
- `TOKENIZATION_MAP[kind].pattern.fullmatch(value)`

__eq__(*other: object*) → bool

Compare against `other` of the same class based on all the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

tokenize(*text: str*) → List[*Token*]

Tokenize the given `text`.

Ensures

- ```
  not (len(result) > 0)
  or result[0].start == 0
  ```

  (Text tokenized from the start)

- ```
  not (len(result) > 0)
  or result[-1].end == len(text)
  ```

  (Text tokenized till the end)

- ```
  all(
      token1.end == token2.start
      for token1, token2 in common.pairwise(result)
  )
  ```

  (Tokens consecutive)

- ```
  all(
      token.value == text[token.start:token.end]
      for token in result
  )
  ```

  (Token values correct)

- `tokens_to_text(result) == text`

tokens_to_text(*tokens: Sequence[*Token*]*) → str

Serialize the `tokens` back into the original text.

Ensures

- `tokens == tokenize(result)`

class UnOp(*value*)

Represent unary operators.

Attributes:

MINUS	Unary negative

MINUS = '-'

Unary negative

class Associativity(*value*)

Represent the associativity of a binary operator.

Attributes:

LEFT	Left associative
RIGHT	Right associative

LEFT = 'Left'

Left associative

RIGHT = 'Right'

Right associative

class BinOpInfo(*precedence: int*, *associativity:* Associativity)

Specify precedence and associativity.

Methods:

__init__(precedence, associativity)

__init__(*precedence: int*, *associativity:* Associativity) → None

class BinOp(*value*)

Represent binary operators.

Attributes:

ADD	Addition
SUB	Subtraction
MUL	Multiplication
DIV	Division
POW	Power

ADD = '+'

Addition

SUB = '-'

Subtraction

MUL = '*'

Multiplication

DIV = '/'

Division

POW = '^'

Power

IDENTIFIER_RE = re.compile('[a-zA-Z_][a-zA-Z0-9]*')

Express an identifier of a variable or a function.

class Identifier(*value: str*)

Represent an identifier of a variable or of a function.

Methods:

__new__(cls, value)	Enforce the identifier properties on `value`.

static __new__(*cls*, *value: str*) → *Identifier*

Enforce the identifier properties on `value`.

Requires

- `IDENTIFIER_RE.fullmatch(value)`

class Expr

Represent a valid expression as an abstract syntax tree (AST).

class Constant(*value: float*)

Represent a constant in the AST.

Methods:

__init__(value)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*value: float*) → None

Initialize with the given values.

Requires

- `not math.isnan(value)`
- `value >= 0.0`

__eq__(*other: object*) → bool

Compare against `other` of the same class based on the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

class Variable(*identifier:* Identifier)

Represent a variable in the AST.

Methods:

__init__(identifier)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*identifier:* Identifier) → None

Initialize with the given values.

__eq__(*other: object*) → bool

Compare against `other` of the same class based on the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

class UnaryOperation(*target:* Expr, *operator:* UnOp)

Represent an unary operation in the AST.

Methods:

__init__(target, operator)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*target:* Expr, *operator:* UnOp) → None

Initialize with the given values.

__eq__(*other: object*) → bool

Compare against `other` of the same class based on the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

class BinaryOperation(*left:* Expr, *operator:* BinOp, *right:* Expr)

Represent a binary operation in the AST.

Methods:

__init__(left, operator, right)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*left:* Expr, *operator:* BinOp, *right:* Expr) → None

Initialize with the given values.

__eq__(*other: object*) → bool

Compare against `other` of the same class based on the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

class Call(*name: str*, *argument:* Expr)

Represent a function call in the AST.

Methods:

__init__(name, argument)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*name: str*, *argument:* Expr) → None

Initialize with the given values.

Requires

- `re.fullmatch(r"(sin|cos|tan)", name)`

__eq__(*other: object*) → bool

Compare against `other` of the same class based on the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

class TokensWoWhitespace(*tokens: Sequence[*Token*]*)

Represent tokens without whitespace.

Methods:

__new__(cls, tokens)	Enforce the properties on `tokens`.
__getitem__()	Get the token(s) at the given index.
__len__()	Return the number of the tokens.
__iter__()	Iterate over the tokens.

static __new__(*cls*, *tokens: Sequence[*Token*]*) → *TokensWoWhitespace*

Enforce the properties on `tokens`.

Requires

- `all(token.kind != TokenKind.WHITESPACE for token in tokens)`

__getitem__(*index: int*) → *Token*

__getitem__(*index: slice*) → *TokensWoWhitespace*

Get the token(s) at the given index.

__len__() → int

Return the number of the tokens.

__iter__() → Iterator[*Token*]

Iterate over the tokens.

parse_tokens(*tokens: Sequence[*Token*]*) → *Expr*

Parse the given tokens into an expression.

unparse(*expr:* Expr) → str

Convert the AST given as `expr` back to the source code as text.

Ensures

- `parse_tokens(tokenize(result)) == expr`

evaluate(*expr:* Expr, *lookup: Mapping[*Identifier, *float]*) → float

Evaluate the given expression `expr` substituting variables with `lookup`.

collect_variables(*expr:* Expr) → Set[*Identifier*]

Go recursively over the expression and collect the variable names.

2.3.12 Solutions to Exercise 12

These are the auto-documented solutions to the Exercise 12 used during the lecture "Introduction to Programming" at ETH Zurich (Switzerland) in Fall 2019.

The text of the exercise is available (in German) at: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2019/0027_Intro/Homework/u12.pdf

Problem 1

Interpret mathematical programs developed in Exercise 11, Problem 1.

Instead of passing in the values for the expression evaluator, the variables are assigned values in the program.

(We exclude the part of the problem related to implementing a REPL as handling the user input/output is out-of-scope of this corpus.)

Classes:

TokenKind(value)	Define the token.
TokenizationRule(kind, pattern)	Define a regular expression which specifies a token.
Token(value, start, end, kind)	Represent a token of the source code.
UnOp(value)	Represent unary operators.
Associativity(value)	Represent the associativity of a binary operator.
BinOpInfo(precedence, associativity)	Specify precedence and associativity.
BinOp(value)	Represent binary operators.
Identifier(value)	Represent an identifier of a variable or of a function.
Node()	Represent a node of an abstract syntax tree (AST) of a program.
Expr()	Represent a mathematical expression in an AST.
Constant(value)	Represent a constant in AST.
Variable(identifier)	Represent a variable in the AST.
UnaryOperation(target, operator)	Represent an unary operation in the AST.
BinaryOperation(left, operator, right)	Represent a binary operation in the AST.
Function(value)	Enumerate the available functions.
Call(function, argument)	Represent a function call in the AST.
Statement()	Represent a statement in the AST.
Assign(target, expr)	Represent an assignment statement.
Program(body)	Represent a parsed program.
TokensWoWhitespace(tokens)	Represent tokens without whitespace.

Data:

TOKENIZATION	Define rules so that we can map token kind regular expression.
TOKENIZATION_MAP	Map token kind rule to be matched for that token kind.
IDENTIFIER_RE	Express an identifier of a variable or a function.

Functions:

tokenize(text)	Tokenize the given `text`.
tokens_to_text(tokens)	Serialize the `tokens` back into the original text.
parse_program(tokens)	Parse the given tokens into an abstract syntax tree of a program.
unparse(program)	Convert the AST back to the source code.
interpret(program)	Interpret the given program and return the values of the assigned variables.

class TokenKind(*value*)

Define the token.

Attributes:

NUM	Number literal
VAR	Variable (or function) identifier
OP	Operator
OPEN	Opening parenthesis
CLOSE	Closing parenthesis
WHITESPACE	Whitespace (including tabs *etc.*)
ASSIGN	Assignment
SEMICOLON	Semicolon

NUM = 1

Number literal

VAR = 2

Variable (or function) identifier

OP = 4

Operator

OPEN = 5

Opening parenthesis

CLOSE = 6

Closing parenthesis

WHITESPACE = 7

Whitespace (including tabs *etc.*)

ASSIGN = 8

Assignment

SEMICOLON = 9

Semicolon

class TokenizationRule(*kind:* TokenKind, *pattern: Pattern[str]*)

Define a regular expression which specifies a token.

Methods:

__init__(kind, pattern)	Initialize with the given values.
__repr__()	Represent the instance as a string for debugging.

__init__(*kind:* TokenKind, *pattern: Pattern[str]*) → None

Initialize with the given values.

__repr__() → str

Represent the instance as a string for debugging.

TOKENIZATION = [TokenizationRule(1,
re.compile('(inf|(0|[1-9][0-9]*)(\\.[0-9]+)?(e[+\\-]?[0-9]+)?)')), TokenizationRule(2,
re.compile('[a-zA-Z_][a-zA-Z_0-9]*')), TokenizationRule(4, re.compile('[+\\-*/^]')),
TokenizationRule(5, re.compile('\\(')), TokenizationRule(6, re.compile('\\)')),
TokenizationRule(8, re.compile('=')), TokenizationRule(9, re.compile(';')),
TokenizationRule(7, re.compile('\\s+', re.MULTILINE))]

Define rules so that we can map token kind regular expression.

TOKENIZATION_MAP: Mapping[*TokenKind*, *TokenizationRule*] = {<TokenKind.NUM: 1>:
TokenizationRule(1, re.compile('(inf|(0|[1-9][0-9]*)(\\.[0-9]+)?(e[+\\-]?[0-9]+)?)')),
<TokenKind.VAR: 2>: TokenizationRule(2, re.compile('[a-zA-Z_][a-zA-Z_0-9]*')),
<TokenKind.OP: 4>: TokenizationRule(4, re.compile('[+\\-*/^]')), <TokenKind.OPEN: 5>:
TokenizationRule(5, re.compile('\\(')), <TokenKind.CLOSE: 6>: TokenizationRule(6,
re.compile('\\)')), <TokenKind.ASSIGN: 8>: TokenizationRule(8, re.compile('=')),
<TokenKind.SEMICOLON: 9>: TokenizationRule(9, re.compile(';')), <TokenKind.WHITESPACE:
7>: TokenizationRule(7, re.compile('\\s+', re.MULTILINE))}

Map token kind rule to be matched for that token kind.

class Token(*value: str*, *start: int*, *end: int*, *kind:* TokenKind)

Represent a token of the source code.

Methods:

__init__(value, start, end, kind)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on all the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*value: str*, *start: int*, *end: int*, *kind:* TokenKind) → None

Initialize with the given values.

Requires

- `start < end`
- `len(value) == end - start`
- `TOKENIZATION_MAP[kind].pattern.fullmatch(value)`

__eq__(*other: object*) → bool

Compare against `other` of the same class based on all the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

tokenize(*text: str*) → List[*Token*]

Tokenize the given `text`.

Ensures

- ```
  not (len(result) > 0)
  or result[0].start == 0
  ```

  (Text tokenized from the start)

- ```
  not (len(result) > 0)
  or result[-1].end == len(text)
  ```

  (Text tokenized till the end)

- ```
  all(
      token1.end == token2.start
      for token1, token2 in common.pairwise(result)
  )
  ```

  (Tokens consecutive)

- ```
  all(
      token.value == text[token.start:token.end]
      for token in result
  )
  ```

  (Token values correct)

- `tokens_to_text(result) == text`

tokens_to_text(*tokens: Sequence[*Token*]*) → str

Serialize the `tokens` back into the original text.

Ensures

- `tokens == tokenize(result)`

class UnOp(*value*)

Represent unary operators.

Attributes:

MINUS	Unary negative

MINUS = '-'

Unary negative

class Associativity(*value*)

Represent the associativity of a binary operator.

Attributes:

LEFT	Left associative
RIGHT	Right associative

LEFT = 'Left'

Left associative

RIGHT = 'Right'

Right associative

class BinOpInfo(*precedence: int*, *associativity:* Associativity)

Specify precedence and associativity.

Methods:

__init__(precedence, associativity)	Initialize with the given values.

__init__(*precedence: int*, *associativity:* Associativity) → None

Initialize with the given values.

class BinOp(*value*)

Represent binary operators.

Attributes:

ADD	Addition
SUB	Subtraction
MUL	Multiplication
DIV	Division
POW	Power

ADD = '+'

Addition

SUB = '-'

Subtraction

MUL = '*'

Multiplication

DIV = '/'

Division

POW = '^'

Power

IDENTIFIER_RE = re.compile('[a-zA-Z_][a-zA-Z0-9]*')

Express an identifier of a variable or a function.

class Identifier(*value: str*)

Represent an identifier of a variable or of a function.

Methods:

__new__(cls, value)	Enforce the identifier properties on `value`.

static __new__(*cls*, *value: str*) → *Identifier*

Enforce the identifier properties on `value`.

Requires

- `IDENTIFIER_RE.fullmatch(value)`

class Node

Represent a node of an abstract syntax tree (AST) of a program.

Methods:

__repr__()	Represent the instance as a string for debugging.

abstract __repr__() → str

Represent the instance as a string for debugging.

class Expr

Represent a mathematical expression in an AST.

Methods:

__repr__()	Represent the instance as a string for debugging.

abstract __repr__() → str

Represent the instance as a string for debugging.

class Constant(*value: float*)

Represent a constant in AST.

Methods:

__init__(value)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*value: float*) → None

Initialize with the given values.

Requires

- `not math.isnan(value)`
- `value >= 0.0`

__eq__(*other: object*) → bool

Compare against `other` of the same class based on the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

class Variable(*identifier:* Identifier)

Represent a variable in the AST.

Methods:

__init__(identifier)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*identifier:* Identifier) → None

Initialize with the given values.

__eq__(*other: object*) → bool

Compare against `other` of the same class based on the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

class UnaryOperation(*target:* Expr, *operator:* UnOp)

Represent an unary operation in the AST.

Methods:

__init__(target, operator)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*target:* Expr, *operator:* UnOp) → None

Initialize with the given values.

__eq__(*other: object*) → bool

Compare against `other` of the same class based on the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

class BinaryOperation(*left:* Expr, *operator:* BinOp, *right:* Expr)

Represent a binary operation in the AST.

Methods:

__init__(left, operator, right)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*left:* Expr, *operator:* BinOp, *right:* Expr) → None

Initialize with the given values.

__eq__(*other: object*) → bool

Compare against `other` of the same class based on the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

class Function(*value*)

Enumerate the available functions.

Attributes:

SIN	Sine
COS	Cosine
TAN	Tangent

SIN = 'sin'

Sine

COS = 'cos'

Cosine

TAN = 'tan'

Tangent

class Call(*function:* Function, *argument:* Expr)

Represent a function call in the AST.

Methods:

__init__(function, argument)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*function:* Function, *argument:* Expr) → None

Initialize with the given values.

__eq__(*other: object*) → bool

Compare against `other` of the same class based on the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

class Statement

Represent a statement in the AST.

Methods:

__repr__()	Represent the instance as a string for debugging.

abstract __repr__() → str

Represent the instance as a string for debugging.

class Assign(*target:* Identifier, *expr:* Expr)

Represent an assignment statement.

Methods:

__init__(target, expr)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*target:* Identifier, *expr:* Expr) → None

Initialize with the given values.

__eq__(*other: object*) → bool

Compare against `other` of the same class based on the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

class Program(*body: List[*Statement*]*)

Represent a parsed program.

Methods:

__init__(body)	Initialize with the given values.
__eq__(other)	Compare against `other` of the same class based on the properties.
__repr__()	Represent the instance as a string for debugging.

__init__(*body: List[*Statement*]*) → None

Initialize with the given values.

__eq__(*other: object*) → bool

Compare against `other` of the same class based on the properties.

Otherwise, propagate to `object.__eq__`.

__repr__() → str

Represent the instance as a string for debugging.

class TokensWoWhitespace(*tokens: Sequence[*Token*]*)

Represent tokens without whitespace.

Methods:

__new__(cls, tokens)	Enforce the properties on `tokens`.
__getitem__()	Get the token(s) at the given index.
__len__()	Return the number of the tokens.
__iter__()	Iterate over the tokens.

static __new__(*cls*, *tokens: Sequence[*Token*]*) → *TokensWoWhitespace*

Enforce the properties on `tokens`.

Requires

- `all(token.kind != TokenKind.WHITESPACE for token in tokens)`

__getitem__(*index: int*) → *Token*

__getitem__(*index: slice*) → *TokensWoWhitespace*

Get the token(s) at the given index.

__len__() → int

Return the number of the tokens.

__iter__() → Iterator[*Token*]

Iterate over the tokens.

parse_program(*tokens: Sequence[*Token*]*) → *Program*

Parse the given tokens into an abstract syntax tree of a program.

unparse(*program:* Program) → str

Convert the AST back to the source code.

Ensures

- `parse_program(tokenize(result)) == program`

interpret(*program:* Program) → Mapping[*Identifier*, float]

Interpret the given program and return the values of the assigned variables.

Problem 3

Implement a compiler for the interpreter developed in Exercise 12, Problem 1.

The program should be compiled in a language based on operand stack.

The following operations are supported:

- `CONST c`: push the value `c` on the stack,
- `LOAD v`: load the value of the variable `v` and push it on the stack,
- `STORE v`: pop a value from the stack and store it to the variable `v`,
- `OP {operation}`: pop two values ("left" and "right"), apply the operation and push the result on the stack, and
- `FUNC f`: pop a value from the stack, apply the function `f` on it and push the result on the stack.

Please see page 5 of the exercise for an example.

Classes:

Instruction()	Represent a bytecode instruction.
Const(value)	Push the constant on the stack.
Load(identifier)	Load a variable from the registry and push it on the stack.
Store(identifier)	Pop a value from the stack and store it in the registry.
UnaryOperation(operator)	Pop the value from the stack, apply the operation and push the result.
BinaryOperation(operator)	Pop the two values from the stack, apply the operation and push the result.
Call(function)	Pop the value from the stack, apply the function and push the result.

Functions:

compile_program(program)	Compile the given program into bytecode instructions.
execute(instructions)	Execute the given `instructions`.
compare_against_interpret(program, result)	Compare against the interpreted program.
compile_and_execute(program)	Compile and execute the given `program`.

class Instruction

Represent a bytecode instruction.

class Const(*value: float*)

Push the constant on the stack.

Methods:

__init__(value)	Initialize with the given values.
__repr__()	Represent the instance as a string for debugging.

__init__(*value: float*) → None

Initialize with the given values.

__repr__() → str

Represent the instance as a string for debugging.

class Load(*identifier:* Identifier)

Load a variable from the registry and push it on the stack.

Methods:

__init__(identifier)	Initialize with the given values.
__repr__()	Represent the instance as a string for debugging.

__init__(*identifier:* Identifier) → None

Initialize with the given values.

__repr__() → str

Represent the instance as a string for debugging.

class Store(*identifier:* Identifier)

Pop a value from the stack and store it in the registry.

Methods:

__init__(identifier)	Initialize with the given values.
__repr__()	Represent the instance as a string for debugging.

__init__(*identifier:* Identifier) → None

Initialize with the given values.

__repr__() → str

Represent the instance as a string for debugging.

class UnaryOperation(*operator:* UnOp)

Pop the value from the stack, apply the operation and push the result.

Methods:

__init__(operator)	Initialize with the given values.
__repr__()	Represent the instance as a string for debugging.

__init__(*operator:* UnOp) → None

Initialize with the given values.

__repr__() → str

Represent the instance as a string for debugging.

class BinaryOperation(*operator:* BinOp)

Pop the two values from the stack, apply the operation and push the result.

Methods:

__init__(operator)	Initialize with the given values.
__repr__()	Represent the instance as a string for debugging.

__init__(*operator:* BinOp) → None

Initialize with the given values.

__repr__() → str

Represent the instance as a string for debugging.

class Call(*function:* Function)

Pop the value from the stack, apply the function and push the result.

Methods:

__init__(function)	Initialize with the given values.
__repr__()	Represent the instance as a string for debugging.

__init__(*function:* Function) → None

Initialize with the given values.

__repr__() → str

Represent the instance as a string for debugging.

compile_program(*program:* Program) → List[*Instruction*]

Compile the given program into bytecode instructions.

execute(*instructions: List[*Instruction*]*) → MutableMapping[*Identifier*, float]

Execute the given `instructions`.

Returns

The final state of the program

compare_against_interpret(*program:* Program, *result: Mapping[*Identifier, *float]*) → bool

Compare against the interpreted program.

compile_and_execute(*program:* Program) → MutableMapping[*Identifier*, float]

Compile and execute the given `program`.

Ensures

- `compare_against_interpret(program, result)`

Problem 4

Analyze the words of a text.

We consider not only the content of a word as string, but also its position in the text. A word can appear in multiple locations in the text. (We consider inflections as different words: "go" and "goes" are considered two different words.)

We define an order of the words. A word `x` is smaller than a word `y` if the difference between the first and last occurrence of `x` is smaller than the corresponding difference for `y`.

Provide a function to extract words from a text.

Provide a function `top` to return the largest `n` words based on the aforementioned order.

Data:

WORD_RE	Express a normalized word of a text.
TOKEN_RE	Express a token of a text.

Classes:

Token(text)	Represent a word as a token of the text.
WordOccurrence(first, last, text)	Represent a word occurence in the text.

Functions:

tokens_to_words(tokens)	**ensures**
tokenize(text)	Tokenize the text into normalized word tokens ignoring the punctuation.
find_top(word_occurrences, limit)	Find the `limit` top occurrences in `word_occurrences`.

WORD_RE = re.compile('^[a-z]+(-[a-z])*$')

Express a normalized word of a text.

class Token(*text: str*)

Represent a word as a token of the text.

Methods:

__new__(cls, text)	Enforce the properties on the `text` of the word.

static __new__(*cls*, *text: str*) → *Token*

Enforce the properties on the `text` of the word.

Requires

- `WORD_RE.match(text)`

class WordOccurrence(*first: int*, *last: int*, *text:* Token)

Represent a word occurence in the text.

Methods:

__init__(first, last, text)	Initialize with the given values.
__lt__(other)	Compare against `other` based on the *first* and *last*.
__le__(other)	Compare against `other` based on the *first* and *last*.
__repr__()	Represent the word occurrence as string for easier debugging.

Attributes:

first	Index of the first occurrence
last	Index of the last occurrence
text	Text of the word

__init__(*first: int*, *last: int*, *text:* Token) → None

Initialize with the given values.

Requires

- `last >= 0`
- `first >= 0`
- `first <= last`

first

Index of the first occurrence

last

Index of the last occurrence

text

Text of the word

__lt__(*other:* WordOccurrence) → bool

Compare against `other` based on the *first* and *last*.

__le__(*other:* WordOccurrence) → bool

Compare against `other` based on the *first* and *last*.

__repr__() → str

Represent the word occurrence as string for easier debugging.

tokens_to_words(*tokens: List[*Token*]*) → List[*WordOccurrence*]

Ensures

- `len(tokens) > 0 len(result) > 0`
- `len(result) <= len(tokens)`

- ```
  all(
      tokens[word_occurrence.first] == word_occurrence.text
      and tokens[word_occurrence.last] == word_occurrence.text
      for word_occurrence in result
  )
  ```

- ```
  (
          word_texts := [word_occurrence.text for word_occurrence in␣
  ↪result],
          len(word_texts) == len(set(word_texts))
  )[1]
  ```

  (No duplicate word occurrences)

TOKEN_RE = re.compile('[a-zA-Z]+(-[a-zA-Z])*')

Express a token of a text.

tokenize(*text: str*) → List[*Token*]

Tokenize the text into normalized word tokens ignoring the punctuation.

Ensures

- `sum(len(token) for token in result) <= len(text)`

find_top(*word_occurrences: List[*WordOccurrence*]*, *limit: int*) → List[*WordOccurrence*]

Find the `limit` top occurrences in `word_occurrences`.

Requires

- `limit > 0`

Ensures

- `len(result) == min(len(word_occurrences), limit)`

- ```
  all(
      result[i] >= result[i + 1]
      for i in range(len(result) - 1)
  )
  ```

- ```
  (
          word_set := set(word_occurrences),
          all(
              word_occurrence in word_set  # pylint: disable=used-
  ↪before-assignment
              for word_occurrence in result
          )
  )[1]
  ```

CHAPTER

THREE

RECORDED FAILURES

As we developed the solutions to the exercises, we recorded failures detected by the our tools. These records provide a material for education, further research and initial insights what bugs could be (easily) detected.

3.1 Curated Incorrect Programs

The recorded failures consist of the source codes *at the time* of the failure. This is not really optimal as it is hard to compare the working version of the recorded buggy source code with the final solution.

We re-phrased the buggy code by introducing the bug into the final solution. The re-phrasing should allow the reader to easily differentiate the buggy from the final solution.

The thus curated incorrect programs with bugs can be found at: https://github.com/mristin/python-by-contract-corpus/tree/main/python_by_contract_corpus/incorrect_from_recorded

3.2 Note about the Bias

The mistakes that practically occur during the development are highly biased towards the skill level of the developer as well as his/her familiarity with the testing tools. Since we were very few contributors, and the distribution of problem assignment was skewed towards two contributors rather than random, the current corpus is inevitably highly biased towards the dominant contributors.

Nevertheless, we still think that at least the initial practical insights can be won from the recorded failures despite this bias.

CHAPTER

FOUR

INCORRECT PROGRAMS

The recorded incorrect programs (see *Recorded Failures*), are ill-suited for development of the correctness tools. When developing and evaluating a correctness tool we need to clearly distinguish between expected and collateral bugs, and whether our correctness tool is actually buggy. However, we recorded the failures "on-the-go" while solving the programming problems. Thus the recorded code often blurs the actual cause of errors and diverges substantially from the final solution. Analyzing and tracing errors in such code is expectedly tedious and error-prone.

To provide a better and more precise testbed we converted the recorded failures into minimal changes of final solutions. We analyzed the recorded failures and tried to manually re-introduce the bug into the final solution. That way the incorrect program is minimally different to the apparently correct solution which makes such incorrect code easier to reason about and gives succinct and reduced test cases.

While we could re-introduce the most of the failures, some cases were not suitable for re-introduction. This was either because the recorded case was not informative enough or because the recorded case diverged too much from the final solution to be re-introduced in a meaningful manner. For example, the initial and the final solutions oftentimes pursued different directions as we had to completely change the approach and re-model the problem with different abstractions. Thus capturing a bug helped us during the development for re-modelling, but it was later impossible to re-trace the bug and re-introduce it since *knowing* about the bug changed the solution in such a way that it was avoided (*e.g.*, by using a different class hierarchy or different responsibility domains of the functions).

Additionally, for some exercises, the specification of the problem was not detailed enough, so what appears to be a bug is actually an under-specification. We consequently ignored such failure cases though they represent valid bugs.

The code of the incorrect programs can be found at: https://github.com/mristin/python-by-contract-corpus/tree/main/python_by_contract_corpus/incorrect_from_recorded

The complete list of the recorded failure cases which were ignored can be found at: https://github.com/mristin/python-by-contract-corpus/tree/main/python_by_contract_corpus/incorrect_from_recorded/__init__.py

CHAPTER

FIVE

CONTRIBUTING

5.1 Coordinate First

Before you create a pull request, please create a new issue first to coordinate.

It might be that we are already working on similar exercises, but we haven't made our work visible yet.

5.2 Create a Development Environment

We usually develop in a virtual environment. To create one, change to the root directory of the repository and invoke:

```
python -m venv venv
```

You need to activate it. On *nix (Linux, Mac, *etc.*):

```
source venv/bin/activate
```

and on Windows:

```
venv\Scripts\activate
```

5.3 Install Development Dependencies

Once you activated the virtual environment, you can install the development dependencies using `pip`:

```
pip3 install --editable .[dev]
```

The –editable option is necessary so that all the changes made to the repository are automatically reflected in the virtual environment (see also this StackOverflow question).

5.4 Pre-commit Checks

We provide a battery of pre-commit checks to make the code uniform and consistent across the code base.

We use black to format the code and use the default maximum line length of 88 characters.

The docstrings need to conform to PEP 257. We use Sphinx docstring format to mark special fields (such as function arguments, return values *etc.*). Please annotate your function with type annotations instead of writing the types in the docstring.

To run all pre-commit checks, run from the root directory:

```
python precommit.py
```

You can automatically re-format the code with:

```
python precommit.py --overwrite
```

The pre-commit script also runs as part of our continuous integration pipeline.

5.5 Write Commit Message

We follow Chris Beams' guidelines on commit messages:

1) Separate subject from body with a blank line
2) Limit the subject line to 50 characters
3) Capitalize the subject line
4) Do not end the subject line with a period
5) Use the imperative mood in the subject line
6) Wrap the body at 72 characters
7) Use the body to explain *what* and *why* vs. *how*

CHAPTER

SIX

CONTRIBUTORS

The solutions annotated with the contracts were provided by the following authors (equal contribution, in alphabetical order):

- Lauren De bruyn,
- Marko Ristin, and
- Phillip Schanely.

CHAPTER

SEVEN

CHANGELOG

7.1 0.0.1

- This is the first, kick-off version.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

D

E

J

L

M

R

S

T

U

V

W

X

Y

Z