State Design Pattern

Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

Exercise

Write a function that calculates the sum of an array's values while skipping any instances of the value 13 and the value immediately following 13.

Example (client code):

```
nums1 = [1, 13, 10, 1, 13, 13, 13, 10, 1, 13]
nums2 = [1, 13, 10, 1, 13, 13, 13, 10, 1]
nums3 = [13, 10, 1, 13, 13, 13, 10, 1]


print(f(nums1, 13) == 3)
print(f(nums2, 13) == 3)
print(f(nums3, 13) == 2)
```

Solution (1)

```python
def sum_skip_el(nums, el_to_skip):
  sum_val = 0
  last_el = None
  for el in nums:
    if el != el_to_skip and last_el != el_to_skip:
      sum_val += el
    last_el = el
  return sum_val
```

Solution (2)

```python
def sum_skip_el(nums, el_to_skip):
  sum_val = 0
  for i in range(1, len(nums)):
    if nums[i] != el_to_skip and nums[i - 1] != el_to_skip:
      sum_val += nums[i]
  if nums[0] != el_to_skip:
    sum_val += nums[0]
  return sum_val
```

Solution (3)

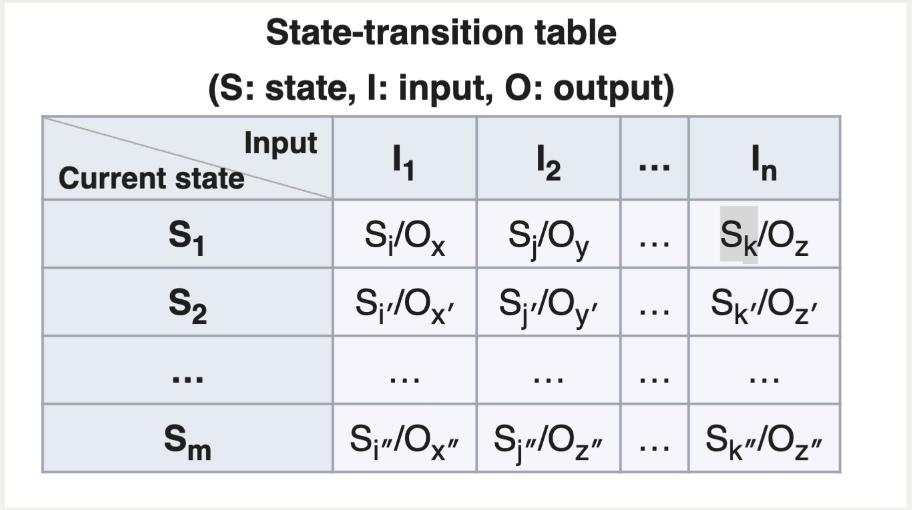
```
def sum_skip_el(nums, el_to_skip):

  def _is_to_sum():  # inner
    if nums[i] == el_to_skip:
      return False
    if i==0:
      return True
    return nums[i-1] != el_to_skip

  sum_val = 0
  for i in range(len(nums)):
    if _is_to_sum():
      sum_val+= nums[i]
  return sum_val
```

The proposed solutions share a common drawback: they are 'ad hoc' solutions.
They do not enable us to define a general criterion for addressing a class of similar problems, where input items are processed sequentially, and the operations performed on a certain item depend on the previous ones.

A **common pattern** is evident in these problems. We encounter an object with a **state**, where the behavior of the object is defined by both the input and the current state. The input triggers a state transition.

We can examine a common approach to address this category of problems.
The concept of state helps us to describe the problem without ambiguity

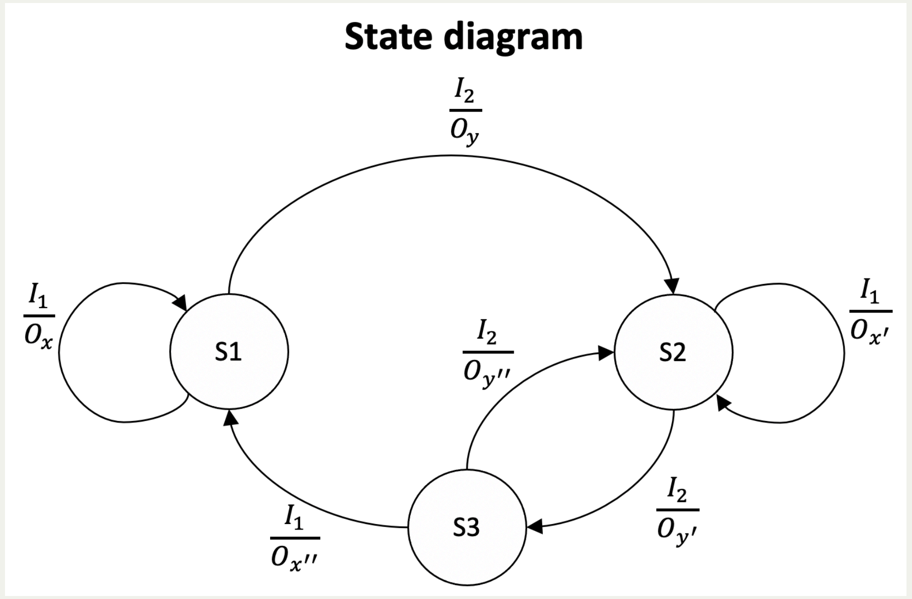
A Finite-state Machine[1] (FSM) is an abstract machine that can be in one of a finite number of states. The FSM can change from one state to another (transition) in response to some inputs.



State-transition table

(S: state, I: input, O: output)

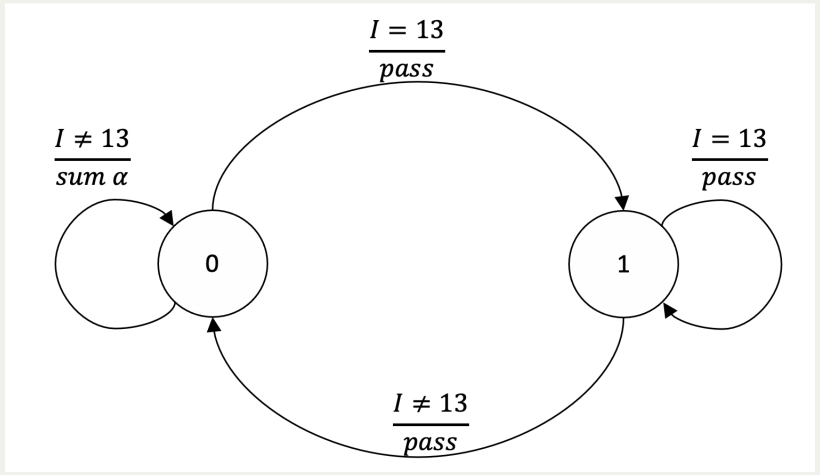
Current state \ Input	I_1	I_2	…	I_n
S_1	S_i/O_x	S_j/O_y	…	S_k/O_z
S_2	$S_{i'}/O_{x'}$	$S_{j'}/O_{y'}$	…	$S_{k'}/O_{z'}$
…	…	…	…	…
S_m	$S_{i''}/O_{x''}$	$S_{j''}/O_{z''}$	…	$S_{k''}/O_{z''}$



[1] https://en.wikipedia.org/wiki/Finite-state_machine. A Mealy machine the output values are determined both by its current state and the current inputs. In a Moore machine the output values are determined solely by its current state. It is also possible to associate actions with a state: an **entry action**, performed when entering the state, and an **exit action**, performed when exiting the state.

We can describe the `sum_skip` exercise (that skip the value 13 and the next element) using the *state* concept:

- The object has two states, 0 and 1.
- In state 0, if the input is $\neq 13$ the object adds the values in input, and the object remains in state 0. If the input is 13, the value is skipped, and the object changes to state 1.
- In state 1, the object does not sum the values in input. If the input is 13, the object remains in state 1. If the input is other than 13, the object changes to state 0 (and do not print the value).



Here is a possible client code, where the **SumSkip** object accepts one element at a time. Other solutions are also viable.

```
nums1 = [1, 13, 10, 1, 13, 13, 13, 10, 1, 13]
nums2 = [1, 13, 10, 1, 13, 13, 13, 10, 1]
nums3 = [13, 10, 1, 13, 13, 13, 10, 1]

list_of_nums = [nums1, nums2, nums3]
list_of_results = [3, 3, 2]
obj = SumSkip(13)

for nums, r in zip(list_of_nums, list_of_results):
  for el in nums:
    obj.sum_skip_el(el)
  print(obj.sum_val == r)
  obj.reset()
```

Write the **SumSkip** class.

Solution

```
class SumSkip:

  def __init__(self, el_to_skip):
    self.el_to_skip = el_to_skip
    self.state = 0
    self.sum_val = 0

  def sum_skip_el(self, el):
    if self.state == 0:
      if el == self.el_to_skip:
        self.state = 1
      else:
        self.sum_val += el
    elif self.state == 1:
      if el == self.el_to_skip:
        pass
      else:
        self.state = 0

  def reset(self):
```

Discussion

In simple cases, explicit conditional logic is often fine.

Problems:

1. The **conditional logic** is prone to errors, particularly when dealing with complex, nested if-elif-else branches.
2. The solution lacks **scalability**. In extensive state machines, the code could span numerous pages of conditional statements. Modifying this code can be challenging and may result in maintenance issues. Simply adding a new state implies altering several functions.
3. **Duplication** is evident. The conditional logic tends to be replicated, with minor variations, in all functions accessing the state variable. This duplication introduces a risk of error-prone maintenance.
4. Lack of **separation of concerns**. There is an absence of clear separation between the transition code of the state machine itself and the actions associated with various events.

A better approach makes use of a **transition table**
(python implementation: the transition table is a dictionary)

- **key**: state
- **value**: a dictionary with:
  - **key**: input
  - **value**: a dictionary containing the action and the next state.

```
transition_table = {

    state_0: {
        input_0: {"action": action, "next_state": next_state},
        input_1: {"action": action, "next_state": next_state},
        default_input: {"action": action, "next_state": next_state}
    },

    state_1: {
        input_0: {"action": action, "next_state": next_state},
        input_1: {"action": action, "next_state": next_state},
        default_input: {"action": action, "next_state": next_state}
    }
}
```

Define the `SkipWithState` class to correspond with the provided client code. The class should include a transition table and two functions, one for the **'sum'** action and the other for the **'do nothing'** action.

```
nums1 = [1, 13, 10, 1, 13, 13, 13, 10, 1, 13]
nums2 = [1, 13, 10, 1, 13, 13, 13, 10, 1]
nums3 = [13, 10, 1, 13, 13, 13, 10, 1]

list_of_nums = [nums1, nums2, nums3]
list_of_results = [3, 3, 2]
obj = SkipWithState(13)

for nums, r in zip(list_of_nums, list_of_results):
  for el in nums:
    obj.process_input(el)
  print(obj.sum_val == r)
  obj.reset()
```

Solution

```python
class SkipWithState:

  def __init__(self, el_to_skip, start_state=0):
    self.el_to_skip = el_to_skip
    self.sum_val = 0
    self.state = start_state
    self.transition_table = {
      0: {  # state 0
        el_to_skip: {"action": self.f_null, "next_state": 1},
        "default_input": {"action": self.f_sum, "next_state": 0},
      },
      1: {  # state 1
        el_to_skip: {"action": self.f_null, "next_state": 1},
        "default_input": {"action": self.f_null, "next_state": 0}
      }
    }

  def f_null(self, v):
    pass
```

The design based on a transition table solves the previous problems.

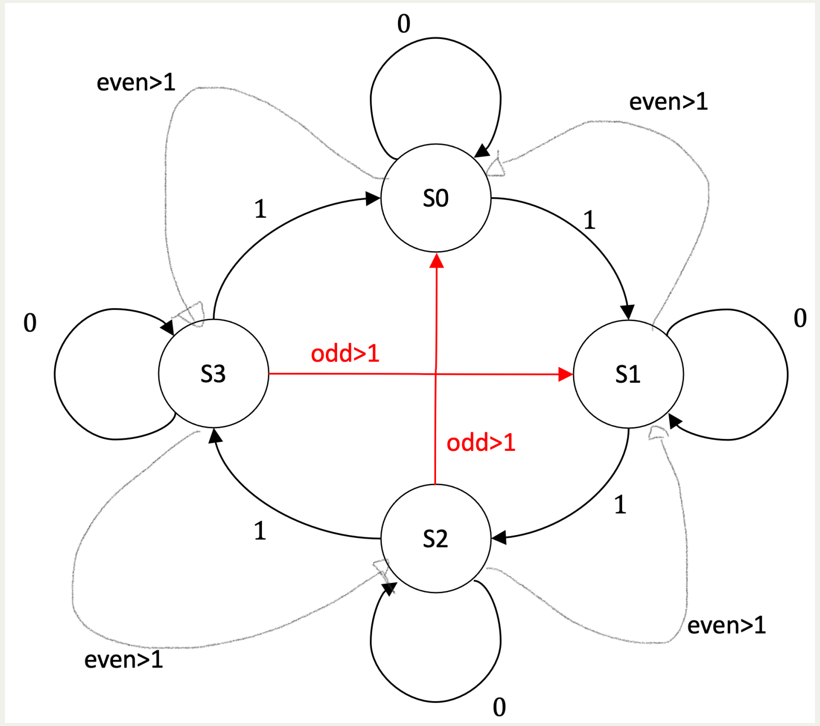
1. It scales well. Independent of the size of the state machine, the code for a state transition is just a lookup-table.
2. No code duplication.
3. Easy to modify. When adding a new state, the change is limited to the transition table.
4. Easy to understand. A well structured transition table provides a good overview of the complete lifecycle.
5. Separation of concerns.

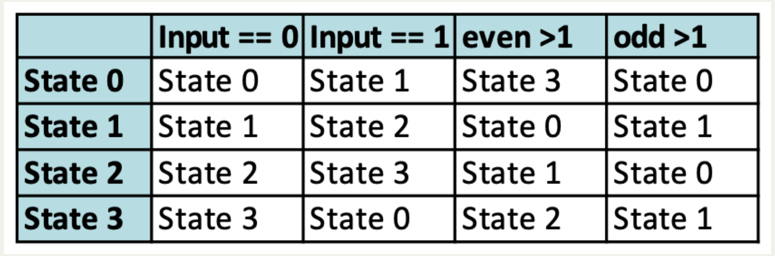
The problem is greatly simplified if each input does not result in actions but **only state transitions**.

The transition table solution is **excellent** if we have **well-defined inputs** and **no actions** to perform after the state transition.

If there are **actions** to be performed after the transition, and these actions may have **different parameters depending on the state and input**, the transition table becomes more complicated, losing some initial elegance and clarity.

What happens with this state machine?





	Input == 0	Input == 1	even >1	odd >1
State 0	State 0	State 1	State 3	State 0
State 1	State 1	State 2	State 0	State 1
State 2	State 2	State 3	State 1	State 0
State 3	State 3	State 0	State 2	State 1

The problem can be solved by 'mapping' the inputs to a series of discrete values (categories).

What if the input categories also depend on the state? The transition table and input management become complicated, losing the initial elegance and clarity.

Conditional structure

`if-elif-else` structure (or switch), one branch for each state

State-transition table

(state, input) $\longrightarrow$ next state

(state, input) $\longrightarrow$ (next state, action)

(state, input) $\longrightarrow$ (next state, action) + default handling

Conditional structure + State-transition table

State-transition table to manage transitions; conditional structures to define actions

Transition tables certainly have their utility, but when actions need to be linked with state transitions and when there isn't a 1-1 correspondence between input values and table entries, the **STATE design pattern** offers a superior alternative.

State Design Pattern

The state design pattern is a way for an object to **change its behavior at runtime**

- according to its internal state and to the input
- **without** using large conditional statements or complicated table lookups.

State-specific code is distributed across different objects rather than localized in a monolithic block.

- It is easier to add actions and states.
- The number of classes makes the code less compact than the other approaches.

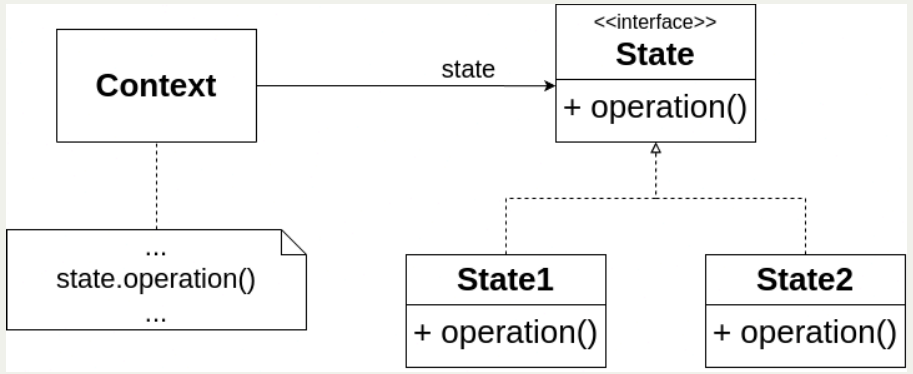
State Design Pattern

The State pattern allows an object to **alter its behavior when its internal state changes**.

- The State pattern define **separate State objects that encapsulate state-specific behavior for each state**.
- An object contains a State object, and **delegates state-specific behavior to its current state object instead of implementing state-specific behavior directly**.
- New states can be added by defining new state classes.
- A class can change its behavior at run-time by changing its current state object.

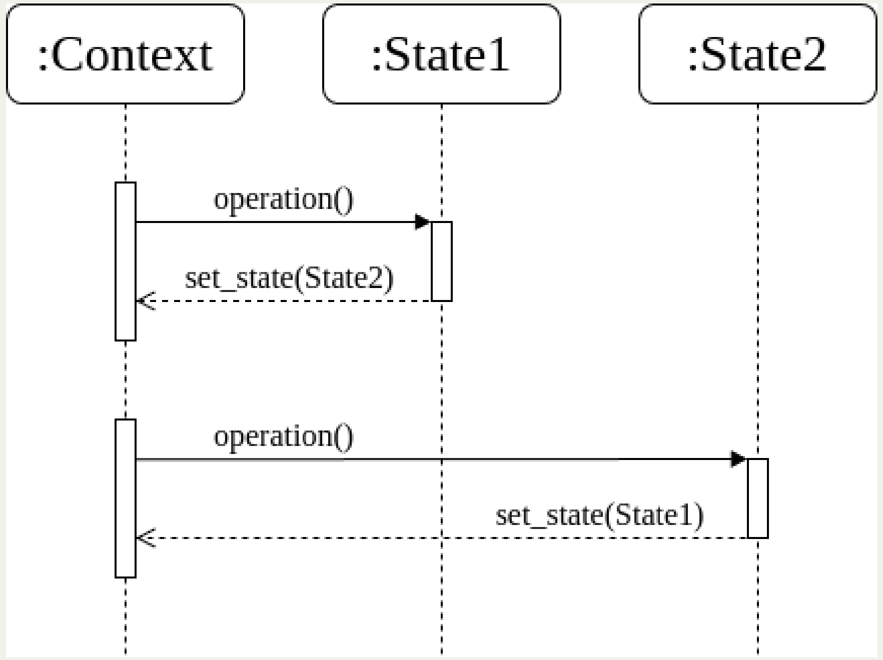
Class diagram

- The Context class (i.e. the object that contains the state) doesn't implement state-specific behavior directly.
- Context refers to the State interface for performing state-specific behavior (`state.operation()`)
- Context is independent of how state-specific behavior is implemented.
- The `State1` and `State2` classes implement the State interface, that is, implement (encapsulate) the state-specific behavior for each state.



The **Sequence diagram** shows the run-time interactions.

- The Context object delegates state-specific behavior to different State objects.
- Context calls `operation()` on its current `State1 object`, which performs the operation and **calls `set_state(State2)` on Context to change context's current state to `State2`**[3].
- Context again calls `operation()` on its current `State2 object`, which performs the operation and **changes context's current state to State1.**



[3] This example shows an object that cyclically changes from state 1 to state 2 and vice versa. It is possible to implement more complex state change logics.

Example

The system has two states (`StateSum` and `StateSkip`) and makes a state transition at each input. The initial state is `StateSum`.

```python
# CLIENT CODE
n = 10
values = list(range(n))
correct_sum = sum(range(0, n, 2))

s = SumSkip()
for el in values:
  s.process_input(el)

print(s.sum_val == correct_sum)
```

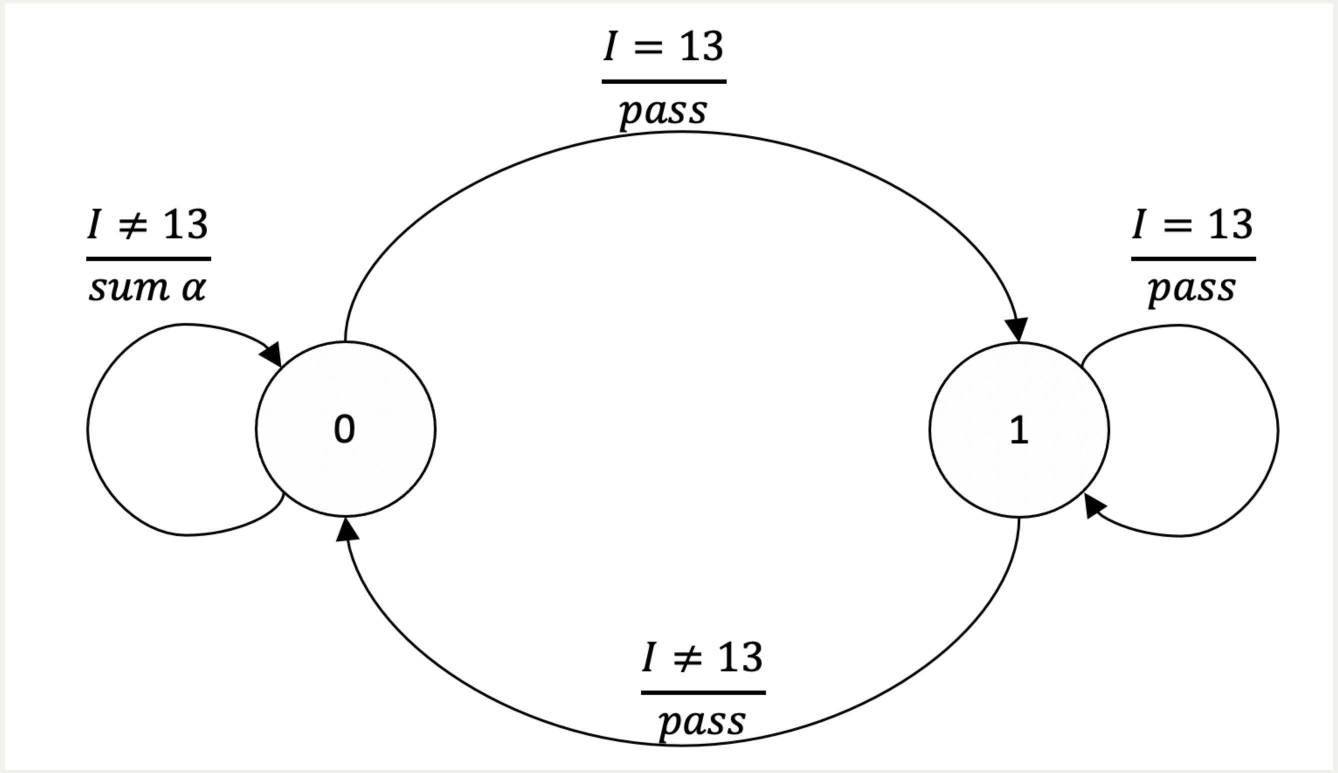
- Create a class for each state, maintaining a consistent interface. One possibility is to define methods like `_action()` and `_change_state()`.
- The `SumSkip` object includes a `State` attribute and delegates all actions to it.

Solution

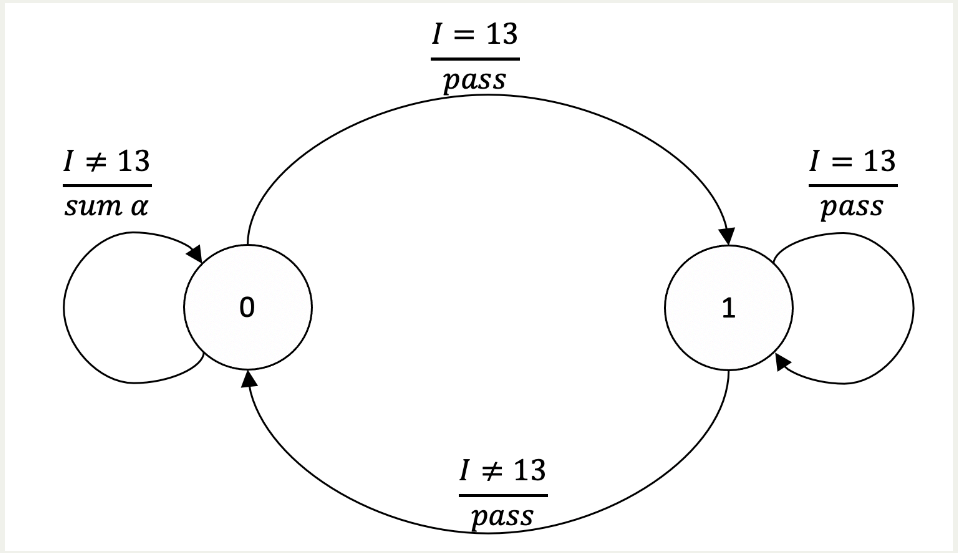
```
from abc import ABC, abstractmethod


class State(ABC):

  def process_input(self, adder, v):
    self._action(adder, v)
    self._change_state(adder, v)

  @abstractmethod
  def _action(self, adder, v):
    pass

  @abstractmethod
  def _change_state(self, adder, v):
  pass


class StateSum(State):
```

Exercise

Solve the original exercise `sum_skip`



```python
# CLIENT CODE
nums1 = [1, 13, 10, 1, 13, 13, 13, 10, 1, 13]
nums2 = [1, 13, 10, 1, 13, 13, 13, 10, 1]
nums3 = [13, 10, 1, 13, 13, 13, 10, 1]

list_of_nums = [nums1, nums2, nums3]
list_of_results = [3, 3, 2]

for nums, r in zip(list_of_nums, list_of_results):
  s = SumSkip(13)
  for el in nums:
    s.process_input(el)
  print(s.sum_val == r)
```



Solution

```
class State(ABC):

  def process_input(self, adder, v):
    self._action(adder, v)
    self._change_state(adder, v)

  @abstractmethod
  def _action(self, adder, v):
    pass

  @abstractmethod
  def _change_state(self, adder, v):
    pass


class StateSum(State):

  def _action(self, adder, v):
    if v != adder.value_to_skip:
      adder.sum_val+= v
```

Write a program that prints a string in **UPPERCASE** or **lowercase** depending on its internal state. The program must print once in uppercase and once in lowercase. In python, the State pattern can be *partially*(4) implemented using first-class function.

```python
def write_lower(printer, name):
  print(name.lower())
  printer.set_state(write_upper)

def write_upper(printer, name):
  print(name.upper())
  printer.set_state(write_lower)


class Printer():

  def __init__(self):
    self.w = write_lower

  def set_state(self, newState):
    self.w = newState

  def write_name(self, name):
     self.w(self,name)
```

```python
p = Printer()

sequence= ["Monday", "Tuesday",
           "Wednesday", "Thursday",
           "Friday", "Saturday", "Sunday"]

for el in sequence:
  p.write_name(el)
```

(4) Here we do not have a real 'state' object. Using this simplified approach is hard to manage complex state transitions.

Exercise

Write a program that prints a **character** in **UPPERCASE** or **lowercase** depending on its internal state.

The program must implement a complex logic, for example:

- it starts to print in lowercase
- it prints in UPPERCASE if it meets the sequence `a`, `b`, `c`
- it returns in the initial state (lowercase) if it meets the character `x`

```
sequence =["a", "a", "b", "c", "d", "e", "f", "x", "a", "d",  "b", "c", "d"]
for c in sequence:
  p.writeChar(c)

# it prints
# a a b c D E F X b a d b c d
```

Suggestion

Instead of using a sequence of characters, use a set of strings. Define a more complex logic with several states and several methods.

Try to add a state. What parts of the code do you need to change?