FRUIZIONE E UTILIZZO DEI MATERIALI DIDATTICI

- **E’ vietata** la **copia**, la **rielaborazione**, la **riproduzione** dei contenuti e immagini presenti nelle lezioni in qualsiasi forma

- **E’ inoltre vietata** la **diffusione**, la **redistribuzione** e la **pubblicazione** dei contenuti e immagini, incluse le registrazioni delle videolezioni con qualsiasi modalità e mezzo non autorizzati espressamente
dall'autore o da Unica

STATE DESIGN PATTERN

Exercise

Write a function that adds the values of an array, skipping all the values equal to 13 and the value immediately after 13.

Example:

```
[1, 13, 10, 1, 13, 13, 13, 10, 1, 13] -> 3
```

Exercise

Write a function that adds the values of an array, skipping all the values equal to 13 and the value immediately after 13.

Example:

```
[1, 13, 10, 1, 13, 13, 13, 10, 1, 13] -> 3
```

Solution: `state/naive_approach/sum_skip_01`

The proposed solutions have a common defect: they are 'ad hoc' solutions. They do not allow us to define a general criterion to solve a class of similar problems.

Let's consider such kind of problems:

- Sum the values in a list, excluding the values between two specific values in the sequence.
  I.e., given the list $[1, 3, 7, \mathbf{10}, \mathbf{10}, 9, 3, 7, \mathbf{10}, 9, 4]$ sum all values skipping those between 7 and 9 in the sequence.

Let's consider such kind of problems:

- Write a function that accepts a list of characters and prints them according to the following rules:
  - the function starts printing lowercase characters
  - when it encounters the **u** character, prints uppercase until it encounters the **l** character
  - characters between two **h**'s are not printed, i.e.,input/output

```
['a','o', 'u', 'm', 'n', 'l', 'o','h', 'x', 'y', 'h', 'a', 'b', 'c']
['a','o',  ? , 'M', 'N', 'l', 'o', ?,            ? , 'a', 'b', 'c']
```

Let's consider such kind of problems:

- A game character gains a superpower if he finds the three amulets **a, b, c** in that order. He can use the superpower three times, then he loses the three amulets and has to start over.
  Variants: he gains superpowers only if he finds the objects **a, b** consecutively, i.e.,

```
x1, a, b, x2, c,...
x1, a, b, c,...
```
are correct sequence.

```
x1, a, x2, b, c,...
```
This is not correct.

It is easy to see a **common pattern** in these problems. We have an **object with a state**.
The behavior of the object depends on the input and the state. The input causes a state transition.

"sum all values skipping those between 7 and 9 in the sequence."

$$[1, 3, 7, \mathbf{10}, \mathbf{10}, 9, 3, 7, \mathbf{10}, 9, 4]$$

It is useful to analyze a possible strategy to solve this class of problems.

It is also useful to better describe these problems, because the presented description is ambiguous.

For example, in problem no. 2, will the characters **u** and **l** be printed, or are they only used to convey commands?
If a **u** character is encountered between the two **h**'s, the object switches to the 'uppercase' state even if it does not print the **u** character?

```
['a', 'o', 'u', 'm', 'n', 'l', 'o','h', 'x', 'y', 'h', 'a', 'b', 'c']
['a', 'o',  ? , 'M', 'N', 'l', 'o', ?,              ? , 'a', 'b', 'c']
```

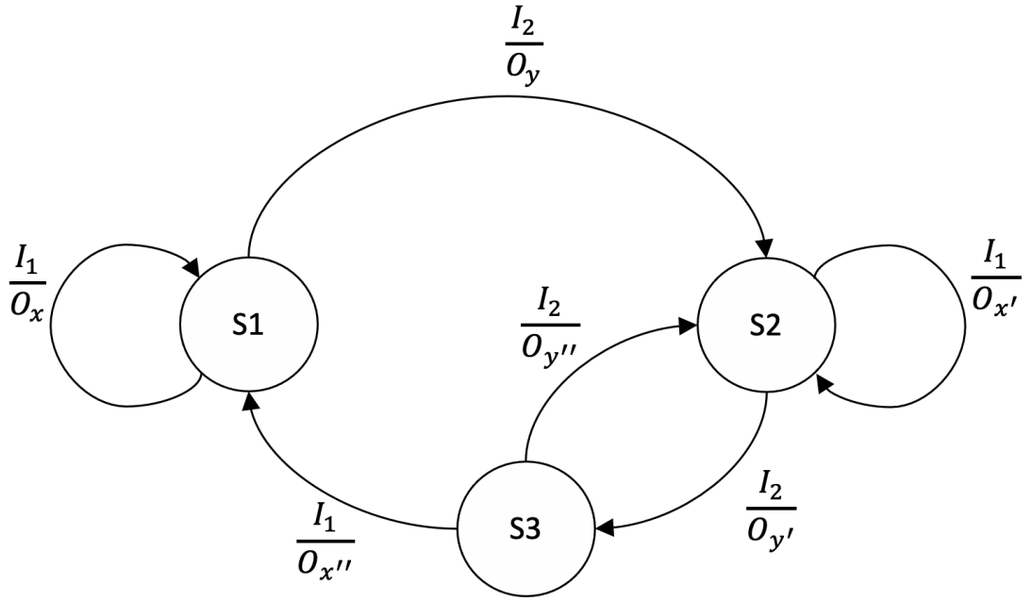
A finite-state machine[1] (FSM) is an abstract machine that can be in one of a finite number of states. The FSM can change from one state to another (transition) in response to some inputs.

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	$S_{i''}/O_{x''}$	$S_{j''}/O_{z''}$	…	$S_{k''}/O_{z''}$

State diagram

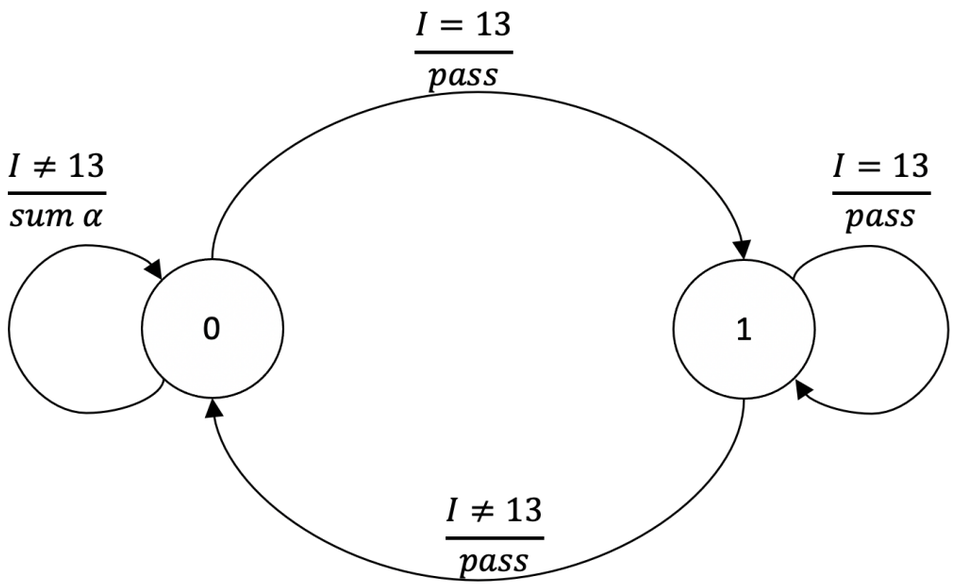


---

[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) in this way (solution: `state/naive_approach/sum_skip_02`. Try to solve it!)

- 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).



The solution in `state/naive_approach/sum_skip_02` uses an integer variable to store the state, and a **conditional structure** to manage the action and the state transition.

You can implement this solution using a function or a class.

Discussion.

In simple cases, explicit conditional logic is often fine.

Problems:

Discussion.

In simple cases, explicit conditional logic is often fine.

Problems:

1. The conditional logic is **error-prone**. It is easy to make errors, especially when there are complex, nested if-elif-else branches.

Discussion.

In simple cases, explicit conditional logic is often fine.

Problems:

1. The conditional logic is **error-prone**. It is easy to make errors, especially when there are complex, nested if-elif-else branches.
2. The solution **doesn't scale**. In large state machines, the code could extend for pages and pages of conditional statements. Applying changes to this code can be very difficult and can lead to maintenance problems. Simply adding a new state implies changing several functions.

Discussion.

In simple cases, explicit conditional logic is often fine.

Problems:

1. The conditional logic is **error-prone**. It is easy to make errors, especially when there are complex, nested if-elif-else branches.
2. The solution **doesn't scale**. In large state machines, the code could extend for pages and pages of conditional statements. Applying changes to this code can be very difficult and can lead to maintenance problems. Simply adding a new state implies changing several functions.
3. **Duplication**. The conditional logic tends to be repeated, with small variations, in all functions that access the state variable. Duplication leads to error-prone maintenance.

Discussion.

In simple cases, explicit conditional logic is often fine.

Problems:

1. The conditional logic is **error-prone**. It is easy to make errors, especially when there are complex, nested if-elif-else branches.
2. The solution **doesn't scale**. In large state machines, the code could extend for pages and pages of conditional statements. Applying changes to this code can be very difficult and can lead to maintenance problems. Simply adding a new state implies changing several functions.
3. **Duplication**. The conditional logic tends to be repeated, with small variations, in all functions that access the state variable. Duplication leads to error-prone maintenance.
4. **No separation of concerns**. There is no clear separation between the code of the state machine itself and the actions associated with the various events.

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

- **key**: tuple (state, input)
- **value**: a dictionary containing the action and the next state.

```
transition_table = {
    (state_0, input_0): {action: action, next_state: next_state},
    (state_0, input_1): {action: action, next_state: next_state},
    # ...
    (state_n, input_k): {action: action, next_state: next_state},
}
```

How to handle cases in which the tuple is not present? For example, it may happen that the input is not explicit.
It is precisely the case of the proposed exercise, in which the input can be $v == 13$ or $v \neq 13$, but all values other than 13 are not explicitly listed.

Another approach makes use of a **transition table** (python implementation: the transition table is a dictionary)
(`state/transition_table/sum_skip`) [(2)]

- **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}
    }}
```

This solution allows us to easily manage the 'default' input using `table.get(key, default)`.

[(2)] version `b` improve version `a` applying the SRP

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.

The solution shown in `state/transition_table/sum_skip` takes advantage of the fact that we have only two possible actions,`f_null()` and `f_sum()`, and that both have the same number of arguments.

However, in the general case, we can have functions with different numbers of arguments, which forces us to use conditional logic again.

```
if (state, input) == (...) :
    f1(a,b)
elif (state, input) == (...) :
    f2(x,y, z, k)
```

If inputs are not explicitly enumerated, it can be hard to represent them in a transition table in an easy way.

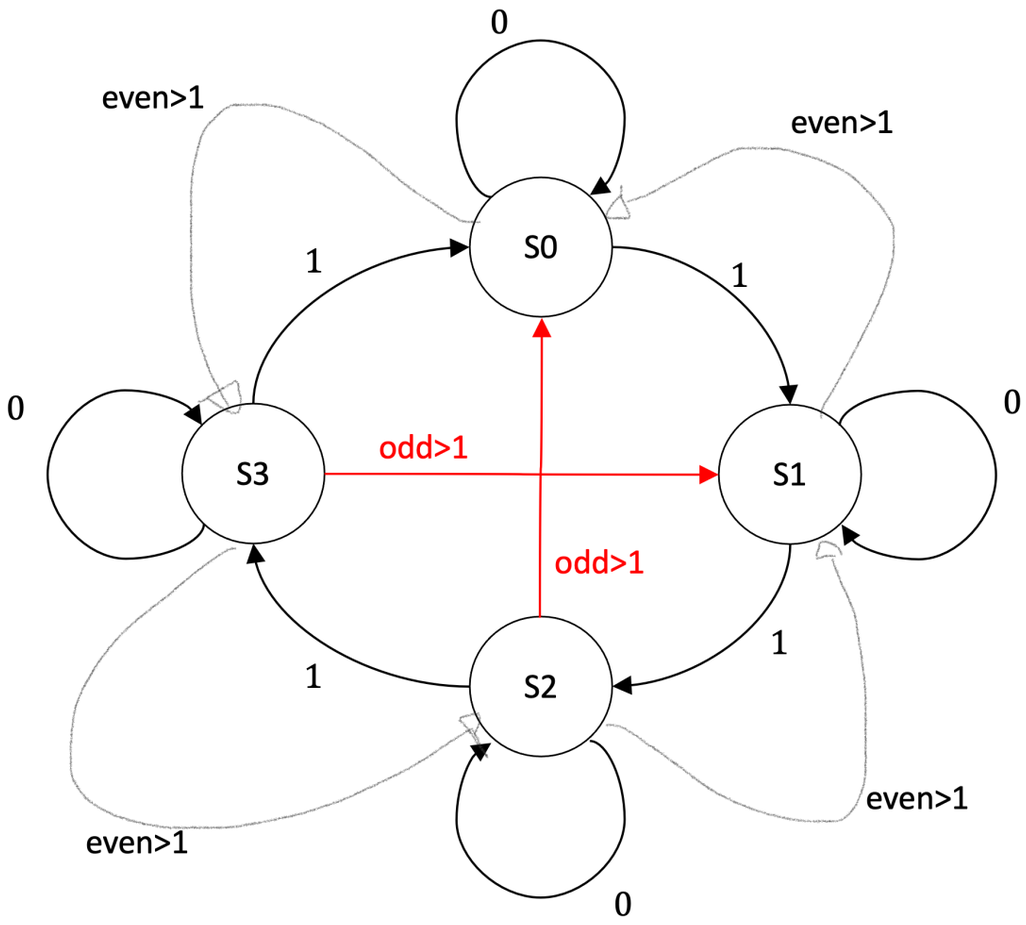
In the previous example we used the feature of python dictionaries to return a default value if the second level key (input) is not present, but it could be a solution that is not always generalizable.

```
dictionary.get(key, default)
```

```
entry_state.get(v_input,
    entry_state['default_input'])
```

	Input == 13	Input ≠ 13
State 0	next_state, *action*	next_state, *action*
State 1	next_state, *action*	next_state, *action*

What happens whit this state machine?

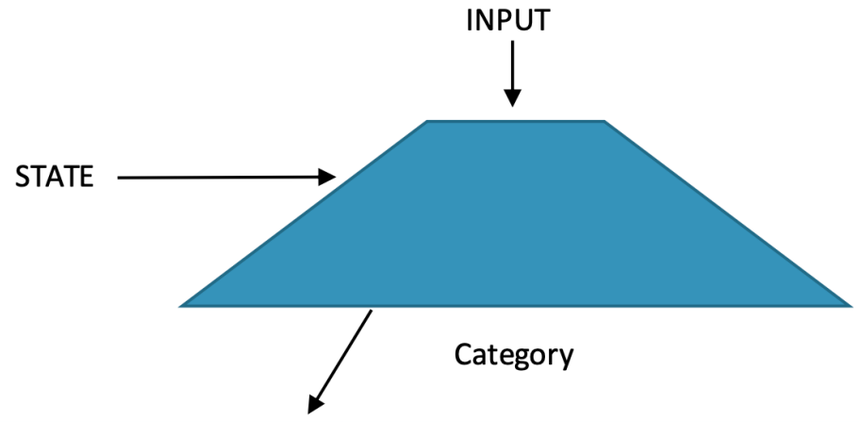


	Input == 0	Input == 1	even >1	odd >1
State 0	State 0	State 1	State 3	State 0
State 1	State 1	State 2	State 0	State 1
State 2	State 2	State 3	State 1	State 0
State 3	State 3	State 0	State 2	State 1

There is no single default input value but two categories (even and odd greater than 1).

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

$$\texttt{input from category } 1 \rightarrow 1$$
$$\texttt{input from category } 2 \rightarrow 2$$
$$\texttt{input from category } n \rightarrow n$$

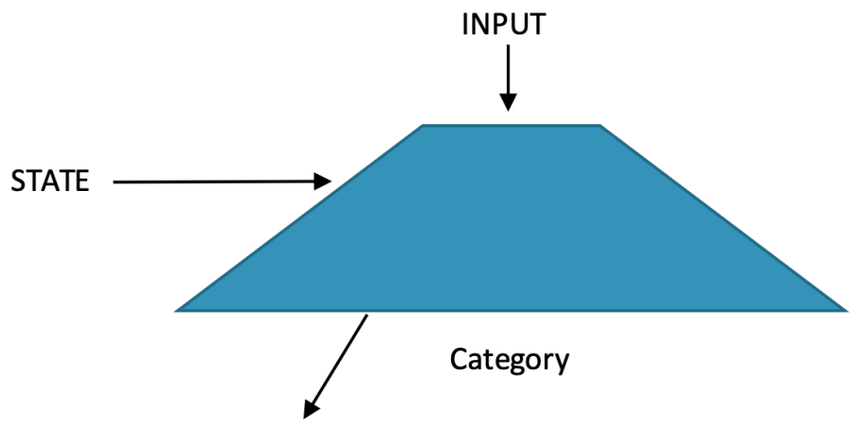


	Category 1	Category 1	Category 3	Category 4
State 0	next_state, action	next_state, action	next_state, action	next_state, action
State 1	next_state, action	next_state, action	next_state, action	next_state, action
State 2	next_state, action	next_state, action	next_state, action	next_state, action
State 3	next_state, action	next_state, action	next_state, action	next_state, action

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

State 0; Categories:

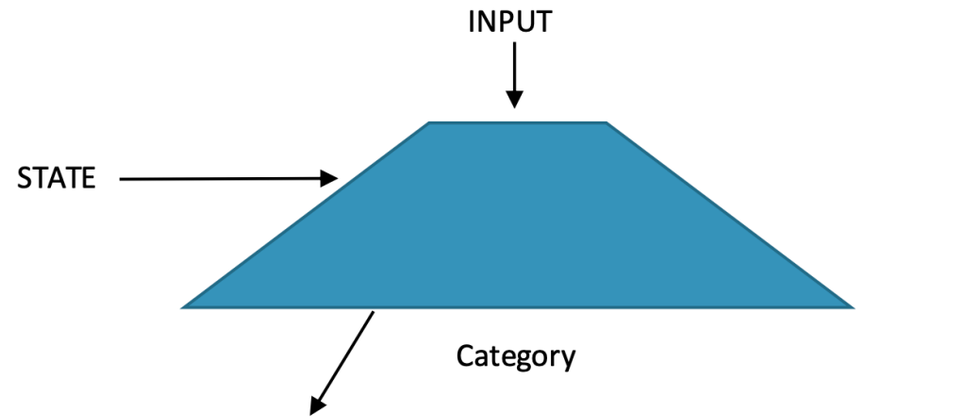
`(input == 0)` ➡ category 1
`(input == 1)` ➡ category 2
`(input >1, odd)` ➡ category 3
`(input >1, even)` ➡ category 4



	Category 1	Category 1	Category 3	Category 4
State 0	next_state, action	next_state, action	next_state, action	next_state, action
State 1	next_state, action	next_state, action	next_state, action	next_state, action
State 2	next_state, action	next_state, action	next_state, action	next_state, action
State 3	next_state, action	next_state, action	next_state, action	next_state, action

State 1; Categories:

`(input moltiple of 3)` ➡ category 5
`(input moltiple of 5)` ➡ category 6
`(input moltiple of 7)` ➡ category 7



	Category 1	Category 1	Category 3	Category 4
State 0	next_state, action	next_state, action	next_state, action	next_state, action
State 1	next_state, action	next_state, action	next_state, action	next_state, action
State 2	next_state, action	next_state, action	next_state, action	next_state, action
State 3	next_state, action	next_state, action	next_state, action	next_state, action

Transition tables definitely have their use, but when actions have to be associated with state transitions, and when there is no 1-1 correspondence between input values and table entry, the **STATE design pattern** provides a better alternative.

STATE DESING 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.

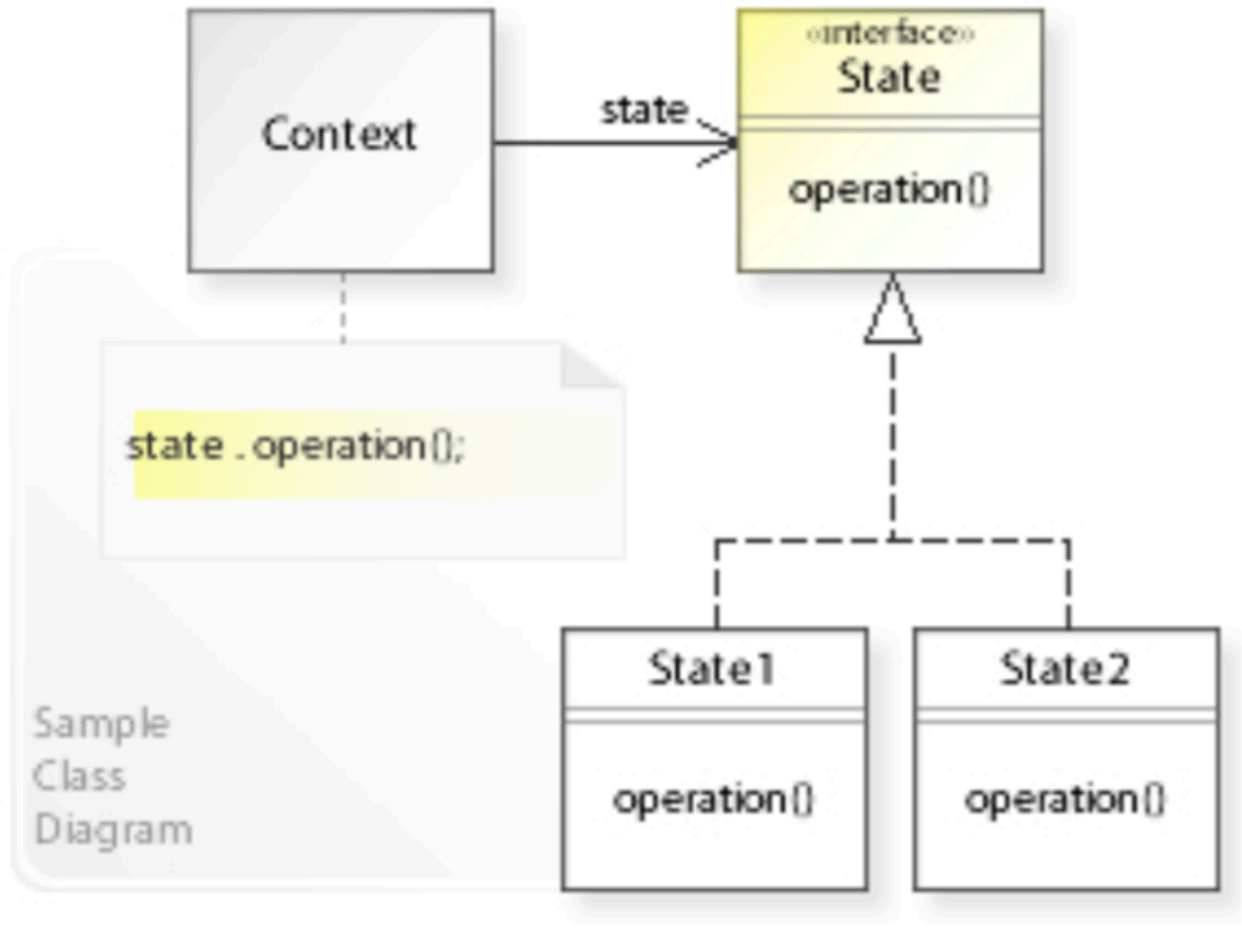
STATE DESING PATTERN

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.

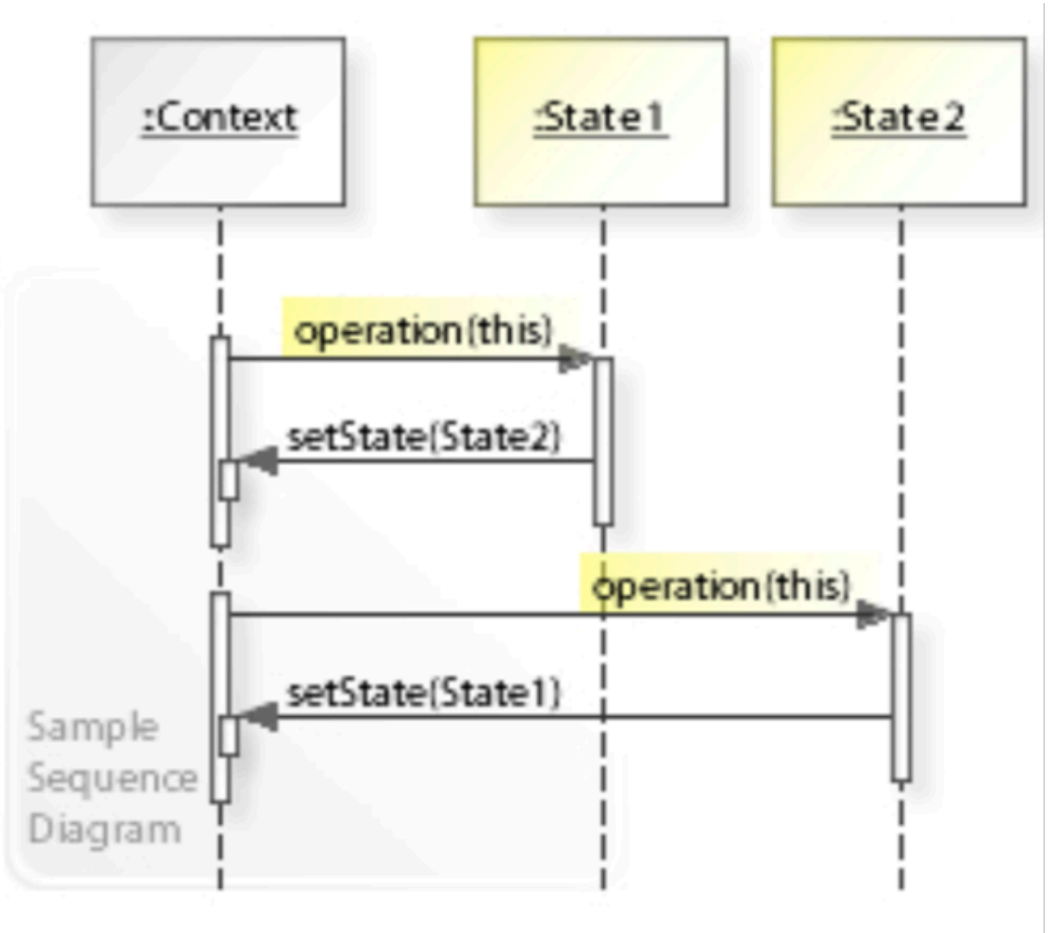
Class diagram

- 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.



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 `setState(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`.

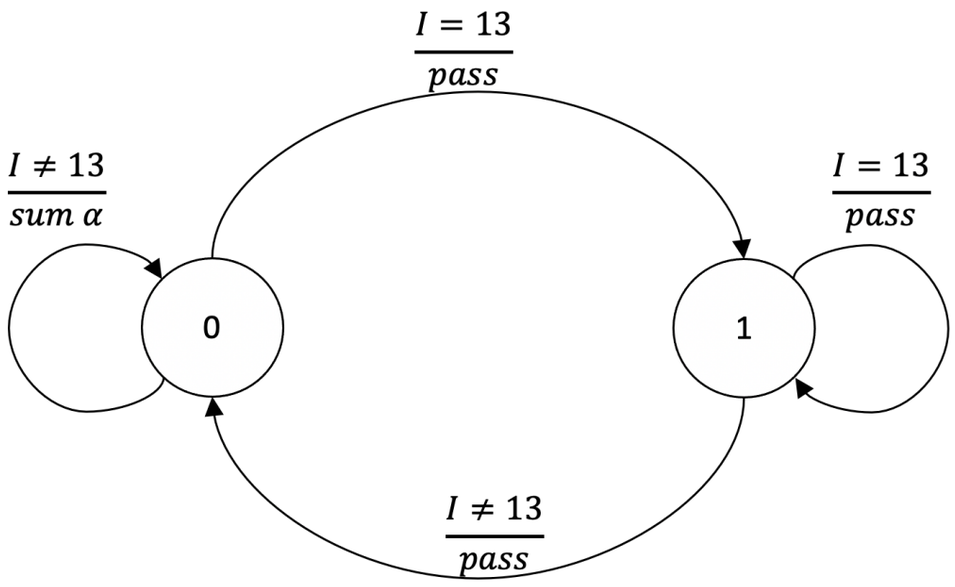
(solution: `state_design_pattern_1`)

Example

Solve the original exercise `sum_skip`

(solution: `state_design_pattern_2`)

Write a function that adds the values of an array, skipping all the values equal to 13 and the value immediately after 13.



[Simplified version] - 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.

```
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)
```

```
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.

Example

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 meet 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
```

Example

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?