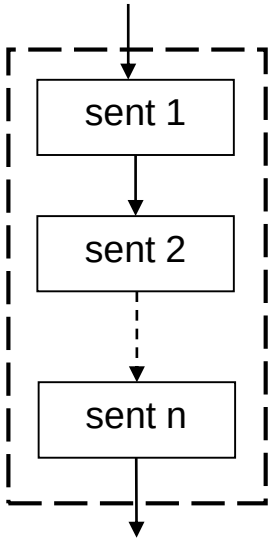
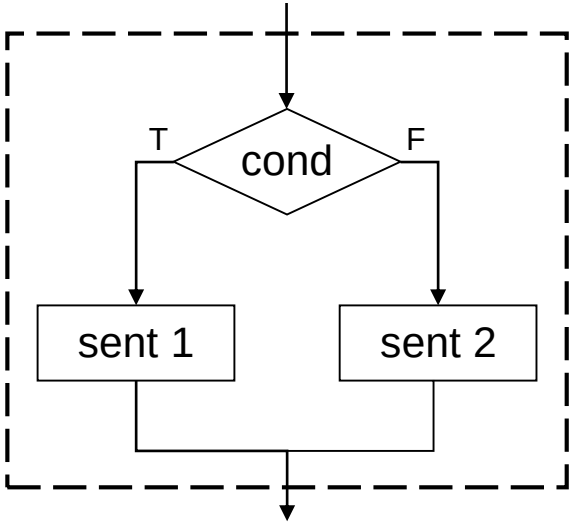
Table of contents

Basic control structures

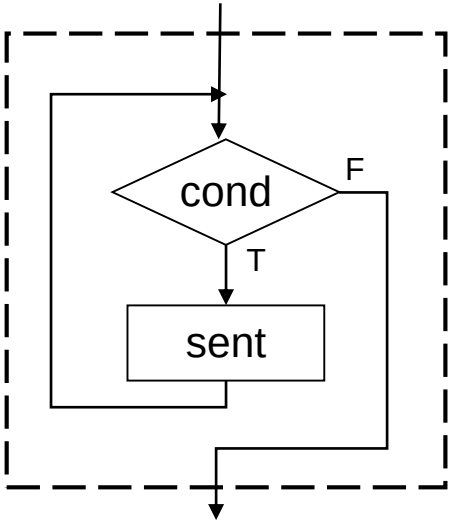
- There are 3 basic control structures:
  - Sequential (**`BLOCK`**)
  - Choice (**`IF-THEN-ELSE`**)
  - Repetitive (**`WHILE`**)



`BLOCK`

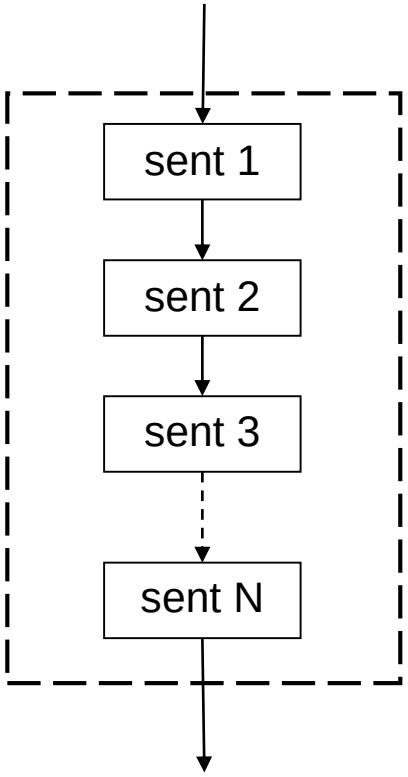


`IF-THEN-ELSE`



`WHILE`

Sequential structure



- It consists in executing one sentence after another.

Implementation

- Several sentences can be grouped to form a **single one**, known as a *composite sentence*.
- Some languages have block delimiters (e.g. braces)
- In Python, what “delimits” a block is to have the same indentation level, i.e. the number of blank spaces on the left.

```
sent1
sent2
sent3
 ...
sentN
```

Sequential structure

- A semicolon can be used to separate sentences if they are on the same line.
- If they are in different lines it is not necessary, but special care with indentation has to be taken.

Examples

Correct

```
sent1
sent2
sent3
sent4
sent5
```

Incorrect

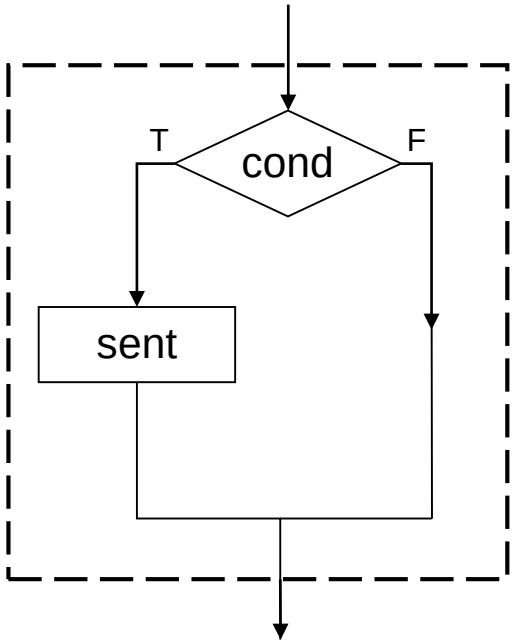
```
sent1          sent1
   sent2     sent2
sent3          sent3
sent4          sent4
sent5          sent5
```

Choice structure

- It allows us to choose between two choices, depending on the result of evaluating a condition (*true* or *false*).

- There are two subtypes
  - Simple choice → `if`
  - Double choice → `if-else`

Simple choice structure (`if`)



- If the “condition” is true, then the “sentence” is executed. If not, no action is executed.

Implementation

```
if cond:
    sent
```

- Note that `sent` is a single sentence.
- If there were more than one sentence, a *composite sentence* has to be formed using the same indentation level.

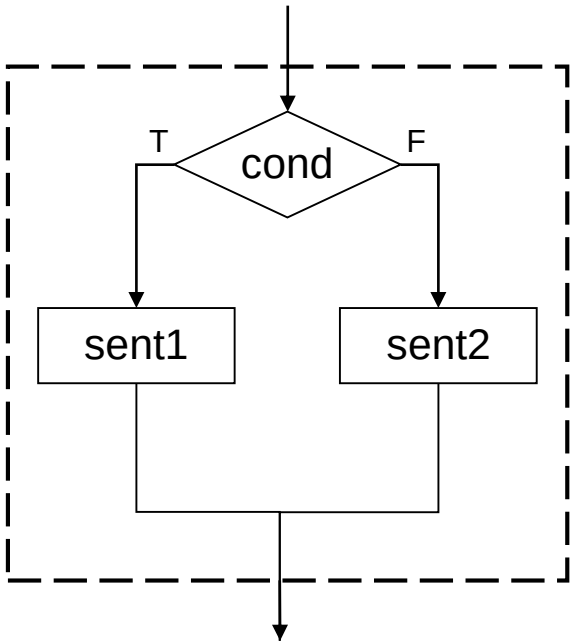
Simple choice structure (`if`)

Example

Assuming that the variable `score` contains the mark of a student, the following fragment of code determines if the test has been passed.

```
# The program gets the score by some means
if score >= 5.0:
    print("Pass")
```

Double choice (`if-else`)



- If the "condition" is *true*, then "sentence #1" is executed. If not, "sentence #2" is executed.

Implementation

```
if cond:
      sent1
else:
      sent2
```

Note that *if* and *else* are aligned

- Both `sent1` and `sent2` are single sentences.
- If there were more than one sentence, a *composite sentence* has to be formed using the same indentation level.

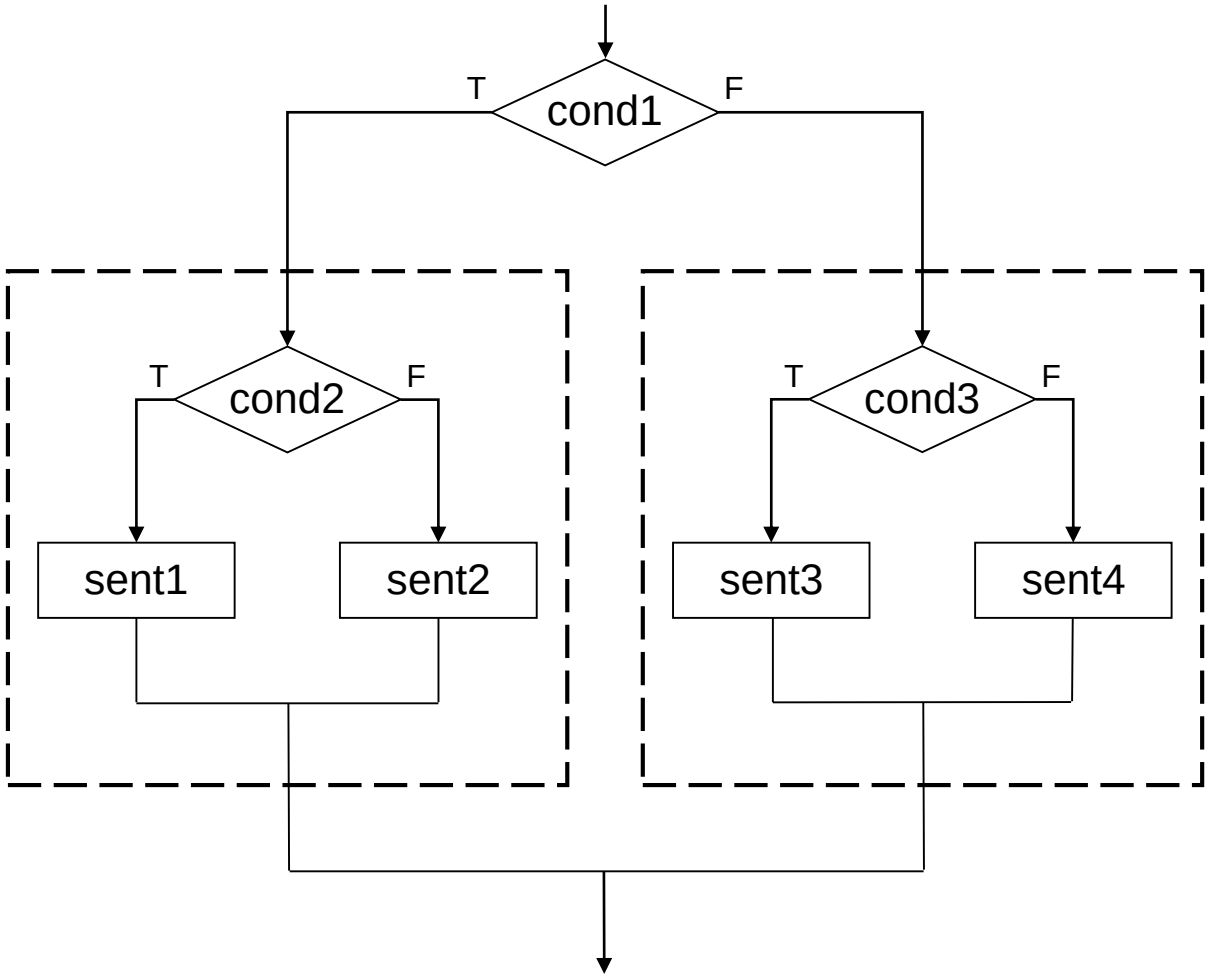
Double choice (`if-else`)

Example

The following fragment of code determines if the `num` integer variable contains an even or odd number.

```
# The num variable is initialized by some means
if num%2 == 0:
  print("The number is even")
else:
  print("The number is odd")
```

Nested choice structures



Implementation

```
if cond1:
    if cond2:
        sent1
    else:
        sent2
else:
    if cond3:
        sent3
    else:
        sent4
```

Multiple choice structure (`if-elif-else`)

- It can be seen as a series of nested `if-else` sentences.
- It allows us to choose among several alternatives:

```
if cond1:
        sent1
elif cond2:
        sent2
elif cond3:
        sent3
...
elif condN:
        sentN
else:
        sentence     #any other case
```

Multiple choice structure (`if-elif-else`)

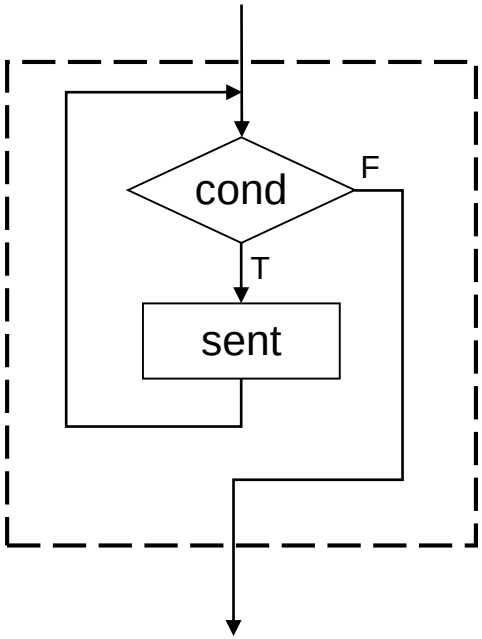
Example

Assuming that the variable `mark` contains the numerical mark of a student, the following fragment of code determines the grading.

```
mark = float(input("Type the exam mark: "))

if (mark >= 0) and (mark < 5):
    print("Fail")
elif (mark >= 5) and (mark < 7):
    print("Pass")
elif (mark >= 7) and (mark < 9):
    print("Very good")
elif (mark >= 9) and (mark <= 10):
    print("Excellent")
else:
    print("Invalid mark")
```

Repetitive structure (`while`)



- Repetitive execution of a sentence (or set of sentences) while the evaluation of a condition is true.
- The condition is evaluated at the beginning → the loop will be executed 0 or more times.

Implementation

```
while cond:
    sent
```

- Note that `sent` is a single sentence.
- If there were more than one sentence, a *composite sentence* has to be formed using the same indentation level.

Repetitive structure (`while`)

Example

The following loops calculates the number of digits of a given integer `n` (standard input).

```
n = int(input("Integer number: "))
a = n
digits = 1

while a//10 != 0:
  digits = digits + 1
  a = a//10

print(n, "has", digits, "digits")
```

Repetitive structure (`while`)

How to build a `while` loop

- Although each program with a loop has a set of variables and a sequence of instructions different, based on the problem to be solved, some general rules can be applied to build a loop (`while`).
- In order to define these rules, we need to know the sequence of **states of the variables** that are generated in its execution step by step.

States of the variables (or status of the program): value of the variables of the problem in each step.

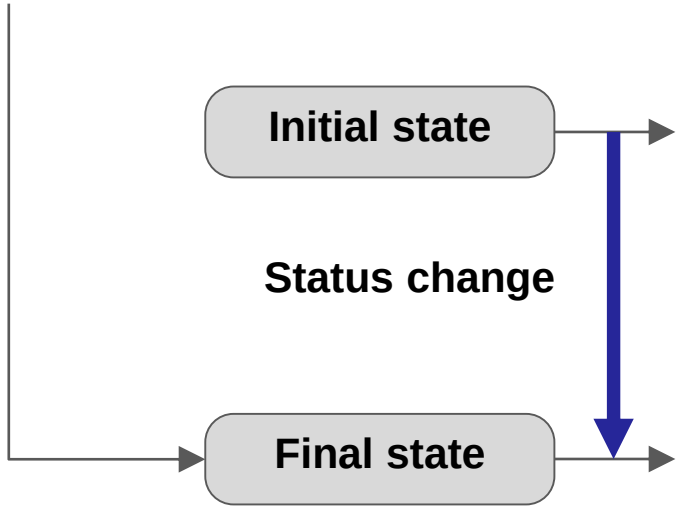
Repetitive structure (`while`)

```
n = int(input("Integer number: "))
a = n
digits = 1                          # block: initial state
while a//10 != 0:
    digits = digits + 1
    a = a //10                      # block: status change
```

Execution step by step

Input:
`n = 62574`

Solution of the problem



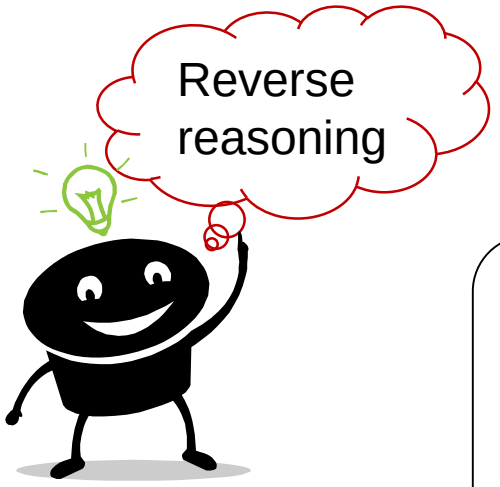
State	Variables			*a//10!=0*
	n	*digits*	*a*	
S_0	62574	1	62574	T
S_1	62574	2	6257	T
S_2	62574	3	625	T
S_3	62574	4	62	T
S_4	62574	5	6	F

Repetitive structure (`while`)

```
# Initial state
initialize_variables
while ¬stop_condition:
    # Status change
    sentence
```

Execution step by step → Sequence of states → **Solution of the problem**

Step 3. Build the loop



Step 1

Find a few initial elements of the sequence of states for the problem to be solved.

Step 2

Determine:

- Initial state
- Stop condition
- Status change

Repetitive structure (`while`)

Rules to build the loop (from step 2 to step 3)

```
# Rule 1: Determine the initial state
initialize_variables     # with input data or assignments
# Rule 2: Determine the stop condition
while ¬stop_condition:
    # Rule 3: Apply the status change
    sentence          # block of sentences needed to
                      # change the status of each variable
```

Final state

- Note that of the variables that form the status of the program, those that contain input data should not be modified.
- For the above example, the `n` variable that contains the integer number typed by the user is not modified.

Repetitive structure (`while`)

- How can we obtain the sequence of initial states? (Step 1):

Understanding and analyzing the problem

allow to

- Determine the status of the program
- Solve the problem for one or more simple cases

Note that the status of a program with a loop is composed of the information (or data stored in variables) needed to change the status.

Repetitive structure (`while`)

Example. Write a program to print on the screen the sequence of integer numbers: `1`, `3`, `6`, `10`, `15`, … that do not exceed a given `threshold` (≥`0`) .

It should not be a problem knowing the next number (`21`),. In that case, we will have determiner how to move from un number to the next one:

`1, 1+2=3, 3+3=6, 6+4=10, 10+5=15, ...`

State	*th*	*number*	*increment*
S_0	20	1	2
S_1	20	1+2=3	3
S_2	20	3+3=6	4
S_3	20	6+4=10	5
S_4	20	10+5=15	6
S_5	20	15+6=21	7

Initial state:

```
th=int(input("Threshold:"))
number=1
increment=2
```

Stop condition:

```
number>th
```

Repetitive structure (`while`)

Status change:

```
print(number, end=" ")
number=number+increment
increment=increment+1
```

Note that in order to change the state, it is necessary to change each variable according to the table.

```
# Initial state
th=int(input("Threshold: "))
number=1
increment=2

# while not stop_condition:
while number <= th:
    # status change
    print(number, end=" ")
    number=number+increment
    increment=increment+1

print()
```

Repetitive structure (`for`)

- The `for` loop is used to repeat an action a certain number of times.
- In this loop structure there is a variable, known as a loop control variable, which takes values between an initial and a final value (sequence of values).
- In Python, these values can be generated by means of the `range()` function.

`range()`

- This function generates a **sequence** of integer numbers
- We will use it with 1, 2, or 3 parameters

Repetitive structure (`for`): `range`

- `range(n)`
  - If `n>0`, it produces an **ascending sequence** of integers, between `0` and `n-1`
  - If `n<=0`, it produces an empty sequence
  - **Example:**

    `range(5)` creates the sequence: `0, 1, 2, 3, 4`

Repetitive structure (`for`): `range`

- `range(m,n)`
  - `m` is the initial value, and `n` is the upper limit
  - If `m<n`, it produces an **ascending sequence** of integers, between `m` and `n-1`
  - If `m>=n`, it produces an empty sequence
  - **Examples:**

    `range(2,5)` creates the sequence: `2, 3, 4`

    `range(-2,3)` creates the sequence: `-2, -1, 0, 1, 2`

  - Note that `range(n)` is equivalent to `range(0,n)`

Repetitive structure (`for`): `range`

- `range(m,n,s)`
  - `m` is the initial value, `n` is the upper limit, and `s` is the step, i.e. the increment (or decrement) between the values generated by the function `range`.
  - If the step `s` is positive, then:
    - If `m<n`, it creates an **ascending sequence** of integers, between `m` and the value prior to `n`, separated by `s`
    - If `m>=n`, it creates an empty sequence
  - If the step `s` is negative, then:
    - If `m>n`, it creates a **descending sequence** of integers, between `m` and the value prior to `n`, separated by `s`
    - If `m<=n`, it creates an empty sequence.
  - **Examples:**

    `range(2,10,2)` creates the sequence: `2, 4, 6, 8`

    `range(10,4,-2)` creates the sequence: `10, 8, 6`

  - Note that `range(n)` is equivalent to `range(0,n,1)`; and `range(m,n)` to `range(m,n,1)`

Repetitive structure (`for`)

Implementation

```
for cv in range(ini, fin, s):
    sent
```

- The loop control variable (`cv`) is initialized to the value `ini`.
- At the end of each iteration, the value of `cv` is modified according to the step `s`.
- The loop ends when `cv` reaches (or exceeds) the value of `fin`, which never appears in the sequence of values

- Note that `sent` is a single sentence.
- If there were more than one sentence, a *composite sentence* has to be formed using the same indentation level.

Repetitive structure (`for`)

Examples

The following loop prints the numbers from 1 to 10 (one per line), it adds them and, at the end, it prints the sum.

```
sum = 0
for i in range(1, 11, 1):
   print(i)
   sum = sum+i
print("Sum:", sum)
```

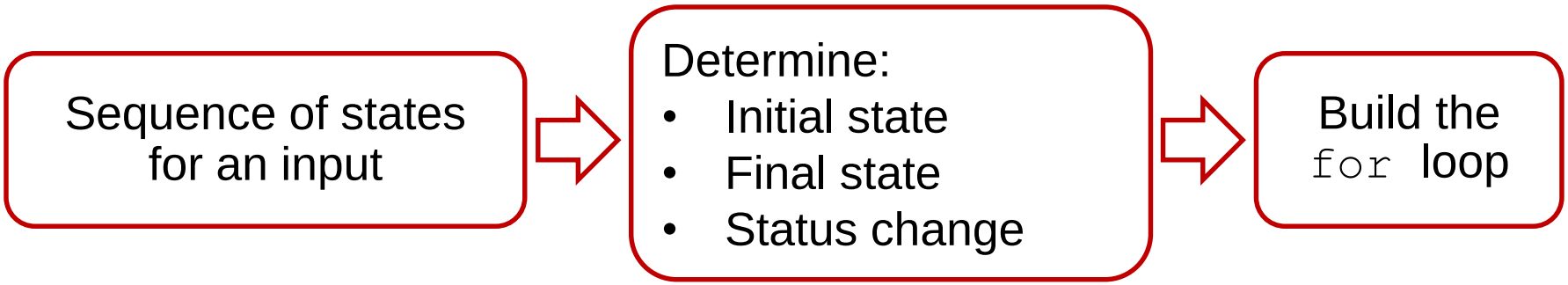
The following loop prints the numbers from 10 to 1 (one per line), in decreasing order.

```
for i in range(10, 0, -1):
   print(i)
```

Repetitive structure (`for`)

How to build a `for` loop

- The procedure is similar to the one explained for the `while` loop.



- Therefore, the rules to build the `for` loop are similar to those explained for the `while` loop. The differences are because of the two main characteristics of this type of loop:

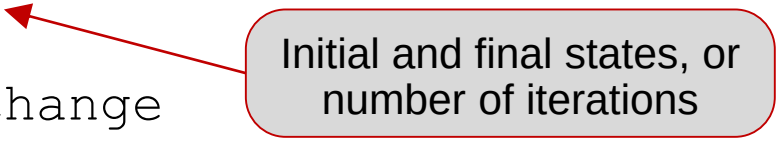
    1. The `for` loop can be only used if we know, *a priori*, the number of iterations (or the initial and finale states of the control variable).

Repetitive structure (`for`)

2. The initialization of variables and their status changes are as for the `while` loop, but for the control variable (`vc`).
   - The loop control variable `cv` is initialized with the function `range`, with a step that determines the status change.

Rules to build the loop

```
# Rule 1: Determine the initial state, but for vc
initialize_variables     # with input data or assignments
# Rule 2: Determine the range of the for loop
for vc in range(ini, fin, step):
    # Rule 3: Apply the status change
    sentence          # block of sentences needed to
                      # change the state of each variable,
                      # but for vc
```



Repetitive structure (`for`)

Example. Given a integer `n` (`n>=0`), write a program that prints on the screen the first `n` numbers of the sequence: `1`, `3`, `6`, `10`, `15`, …

Note that the same example has been used for the `while` loop, so we will use a similar table but changing the input that here is the value of `n`.

In particular, the following table corresponds to the input `n=5`. The variable `increment` is used as the loop variable control.

State	*n*	*number*	*increment*
S_0	5	1	1
S_1	5	1+2=3	2
S_2	5	3+3=6	3
S_3	5	6+4=10	4
S_4	5	10+5=15	5
S_5	5	15+6=21	6

Initial state:

```
n=int(input("Number: "))
number=1
```

Range (as we need `n` numbers, we also need `n` iterations):

```
range(2,n+2,1)
```

Repetitive structure (`for`)

Status change:

```
print(number, end=" ")
number=number+increment
```

Note that in order to change the state, it is necessary to change each variable according to the table, but the control variable.

```
# Initial state
n=int(input("Number: "))
number=1

# range: range(2,n+2,1):
for increment in range(2,n+2,1):
    # status change
    print(number, end=" ")
    number=number+increment

print()
```

Typical appliacionts of loops

Example

Write a program that computes the factorial of a given number.

$$n! = 1 \quad \text{(if } n=0\text{)}$$
$$n! = n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot 2\cdot 1 \quad \text{(if } n>0\text{)}$$

```
n = int(input("Type a number (>=0): "))
while(n<0):
    n = int(input("Type a number (>=0): "))

fact = 1
for i in range(1, n+1):
    fact = fact*i

print (n, "! = ", fact, sep="")
```

Typical application examples

Example

Write a program that computes the semi-factorial of a given number.

$$n!! = 1 \quad \text{(if } n=0\text{)}$$
$$n!! = n \cdot (n-2) \cdot (n-4) \cdot \ldots \cdot 1 \quad \text{(if } n>0\text{)}$$

```
n = int(input("Type a number (>=0): "))
while(n<0):
  n = int(input("Type a number (>=0): "))

semifact = 1
for i in range(n, 0, -2):
  semifact = semifact*i

print (n, "! = ", semifact, sep="")
```