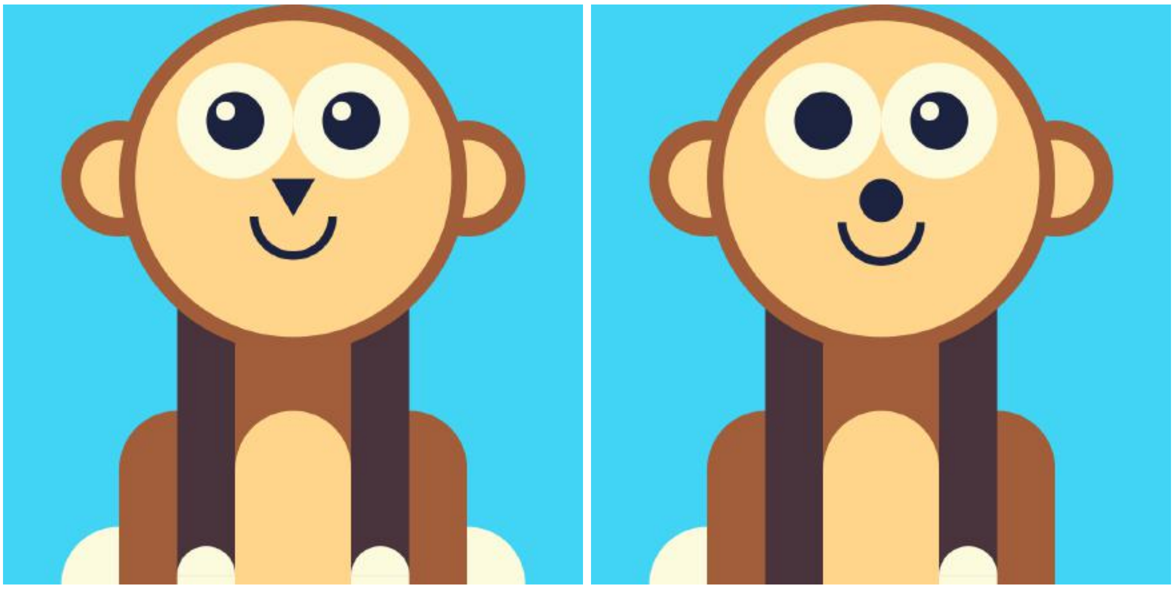
Name:	Date:

Defining Functions

Similarities and Differences

The following two graphics are almost the same. There are many similarities, but they still have some differences. Find them:



In programming, whenever we encounter two elements that look alike but have some differences, we need to **abstract**!

Abstract?

Yes, in programming, to abstract means to replace several pieces of code with a single piece that contains only what is **similar**, and to create a parameter for each difference.

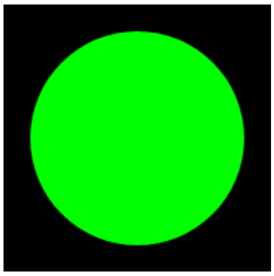
By creating a function I can use one piece of code for multiple purposes.

What is a function?

A function contains the code that is **similar**. For each difference it has a **parameter**.

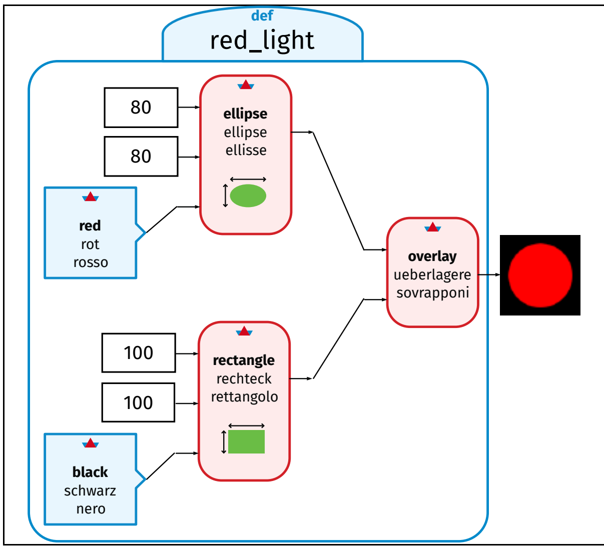
Emergency Traffic Light (A Single Lamp)

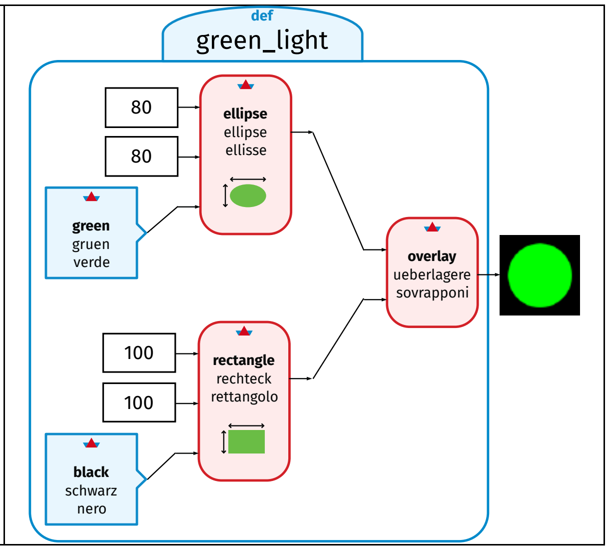
Here are two graphics:



What are their **similarities**?	What are their **differences**?

Let’s use the Tamaro Cards to define a constant for each light. Notice the difference.





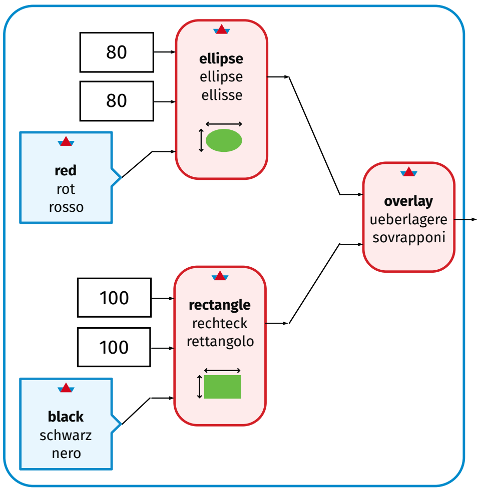
Write the Python codes for the two graphics and then **indicate** their differences:

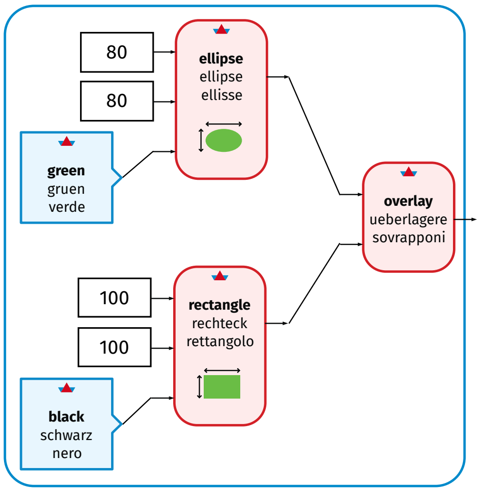
`red_light =`	`green_light =`

Find a good name for each difference:

Perhaps as differences you found the **name** of the constant, and in the program with the Tamaro Cards on the previous page you may have identified the **returned graphic**.

Let's focus on the differences **inside** the definitions of our **constants**:



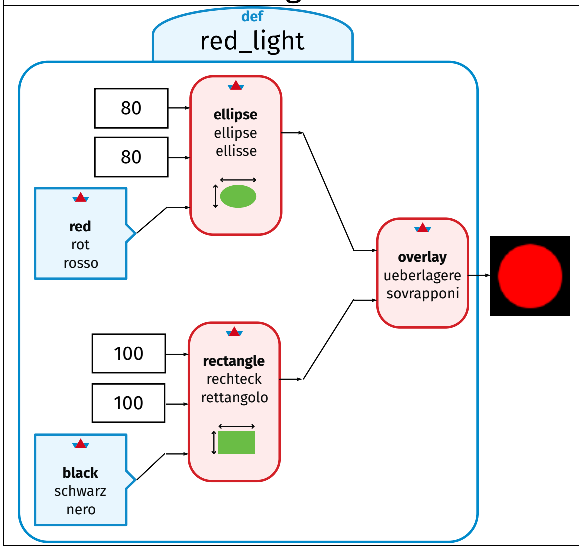
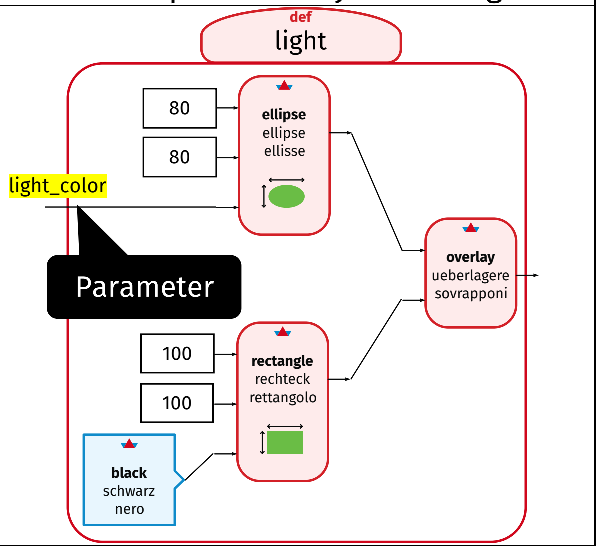


There is only once difference: the color of the ellipse!

Defining a Function

Wouldn't it be nice to be able to write **only one version** of the code, so that it can produce both emergency lights? What about other variations of lights, where only the color changes, for example a blue, magenta, or purple light?

In programming we can do this!
This **abstraction using** parameters is the **super-power** of every programmer!

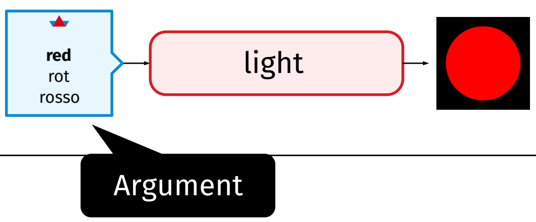
Constant – a red light	**Function** – produce any kind of light
	

The function defined on the right is different from the constant defined on the left: the function has a **parameter** named `light_color` that enters on the left. This parameter will provide the desired color.

Calling a Function

Once a function is **defined**, we can **call** that function to use it.
(This is similar to constants: once defined, we can use them.)

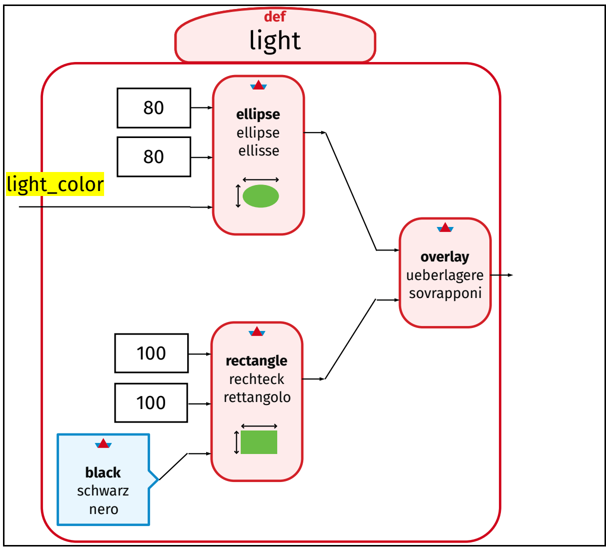
Our `light` function has a **parameter** to specify the color of the emergency light. Thus, we provide the desired color as an **argument** in the function call.

Produce the red light Call of the `light` function with the color `red` as argument:	**Produce the green light** Call of the `light` function with the color `green` as argument:
	

Defining and Calling of Functions in Python

Translation from Tamaro Cards to Python code.

First, we have to write the code to define the `light` function:



Define the parameter

Use the parameter

```
def light(light_color):
  return overlay(
    ellipse(80, 80, light_color),
    rectangle(100, 100, black)
  )
```

In the Python code, to define a function, we need the word **`def`**, followed by the **name** of the function (`light`), followed by the **name** of the parameter in parentheses (`light_color`) and a colon (:).

Following that line, we find the **body** of the function.

```
def FUNCTION_NAME(PARAMETER_NAME):
  return overlay(
    ellipse(80, 80, light_color),
    rectangle(100, 100, black)
  )
```

Body

The **body** of a function contains “the return statement” (**`return`**), which specifies what a function call produces. The statement contains the code that creates the graphic. The code can **use** the name of the **parameter** to refer to the desired color.

Let’s call our function to produce some lights.
Complete the code:

Produce a red light	Produce a green light	Produce a blue light
`light(red)`	`light(`	

Our `light` function is the same as other functions we have seen in the past, for example the functions `rectangle`, `overlay`, or `rgb_color` from the PyTamaro library.

A Function that Produces a Circle

Write a program that creates a blue circle with diameter 100:

Isn't it a bit annoying to use the `ellipse` function to create a simple circle by specifying the diameter **twice**?

Unfortunately, there is no function in the PyTamaro library to create a circle.

Let's define such a function that produces a circle!

Instead of the code you just wrote, using the `ellipse` function, it could be written like this:

```
circle(100, blue)
```

We only need to define the `circle` function, with two **parameters**:
the `diameter` and the `color`.

Define the `circle` function with Tamaro Cards:

Write the function in Python. In the body of the function, place the return statement (**`return`**) that contains a call to the `ellipse` function:

```
def            (          ,          ):
```

The Traffic Light

Program with Tamaro Cards the `traffic_light` function using the `light` function defined above.

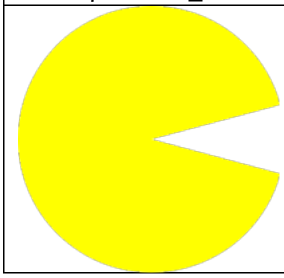
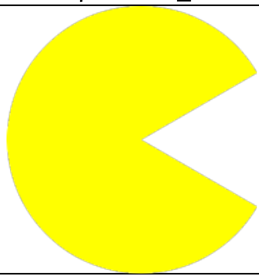
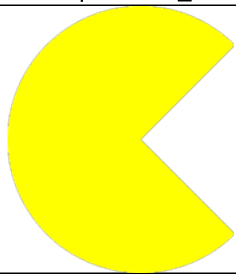
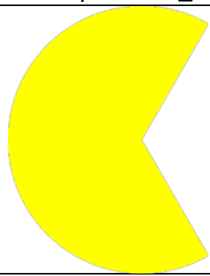
The `traffic_light` function has three parameters:
`traffic_light(color_top, color_middle, color_bottom)`

Tamaro Cards
Python Code

Make the traffic light graphic within the “Semaforo” activity on the PyTamaro site and try out the various color variations!

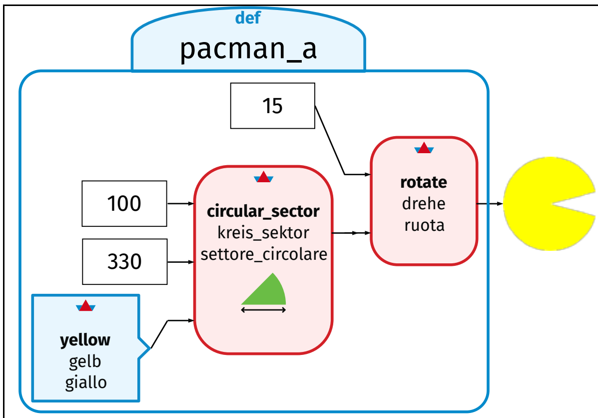
Construction of Different Pac-Men

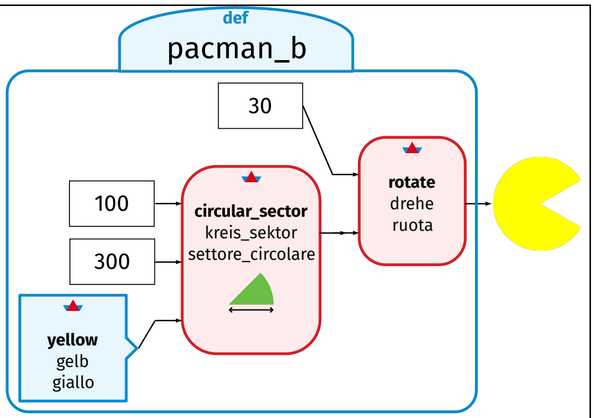
Here are four Pac-Man graphics:

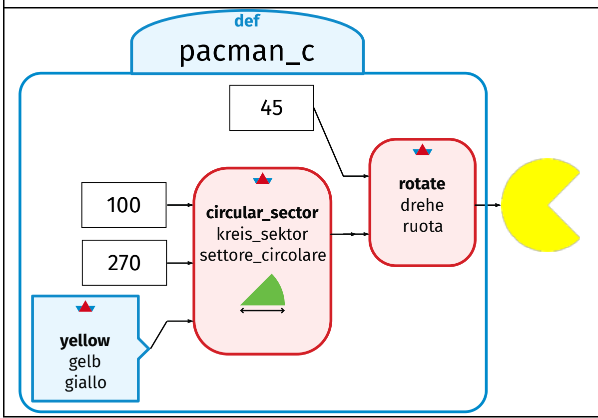
pacman_a	pacman_b	pacman_c	pacman_d
			

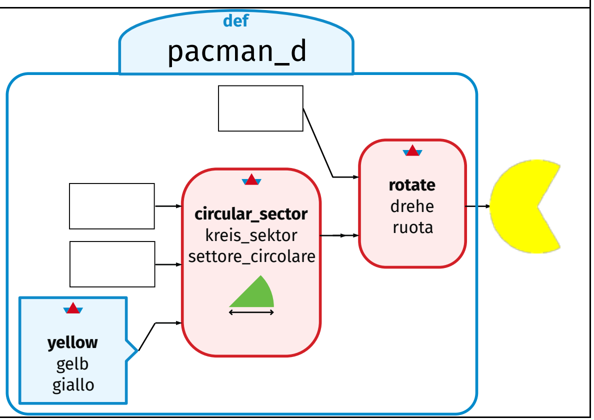
Their **similarities**?	Their **differences**?

We use Tamaro Cards to define four constants, one for each Pac-Man.
Highlight the differences and complete the constant definition for `pacman_d`.









Write the Python code of the first and last graphics and highlight all differences:

`pacman_a =`	`pacman_d =`

Find appropriate names for the differences found.

Did you find two differences?

Complete the following table:

Differences	pacman_a	pacman_b	pacman_c	pacman_d
Rotation angle	15	30	45	
Angle of circular sector	330	300	270	
Mouth angle	*30*	*60*	*90*	*120*

The values for the mouth angle probably don't show up in your code.
The mouth angle is important in creating our Pac-Man, we would like to be able to decide how open it should be!

Let's calculate the rotation angle and the angle of the circular sector for a given mouth angle!

Given the angle of the mouth opening, **write** an expression that produces the rotation angle:
(use the name `mouth_angle` in your expression)

```
rotation_angle =
```

Given the angle of the mouth opening, **write** an expression that produces the angle of the circular sector:
(use the name `mouth_angle` in your expression)

```
sector_angle =
```

Below are the two Python codes for the first and last Pac-Man graphics:

```python
mouth_angle = 30
rotation_angle = mouth_angle / 2
sector_angle = 360 – mouth_angle
pacman = rotate(
  rotation_angle,
  circular_sector(
    100,
    sector_angle,
    yellow
  )
)
```

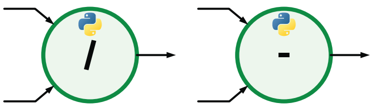
```python
mouth_angle = 120
rotation_angle = mouth_angle / 2
sector_angle = 360 – mouth_angle
pacman = rotate(
  rotation_angle,
  circular_sector(
    100,
    sector_angle,
    yellow
  )
)
```

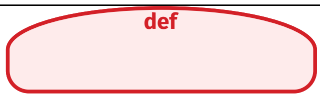
In these two codes, the only difference is the number that defines the mouth-opening angle!

Time to program!

Library

Operators:






```python
def pacman(                    ):
  return
```

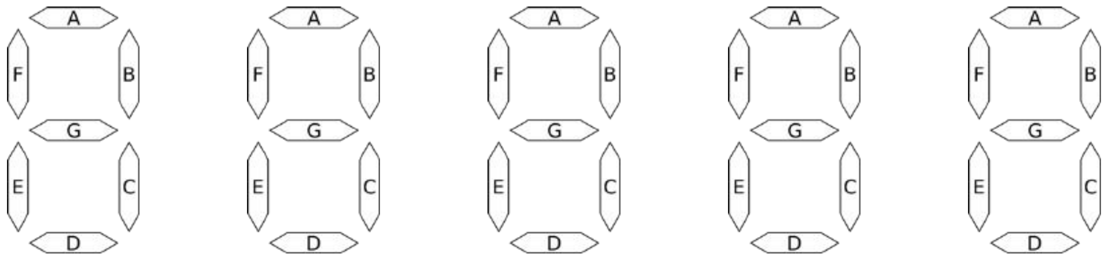
Create your Pac-Man in the "PacMan" activity on the PyTamaro site!

Seven Segment Display

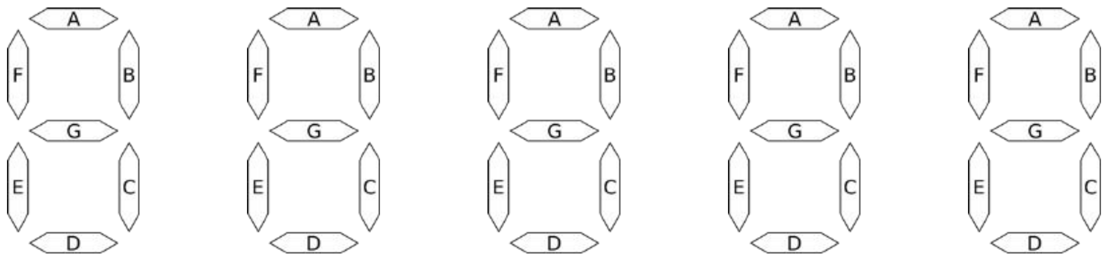
Some old digital clocks and other electronic devices represent a numerical digit (a number from 0 to 9) using a composition of seven LED lights. In the adjacent photo you see such a seven-segment display.

Color the LEDs in the following displays in a way to produce all the decimal digits:

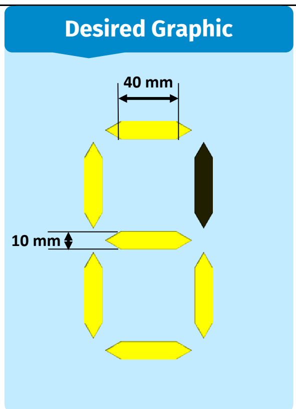
0 1 2 3 4



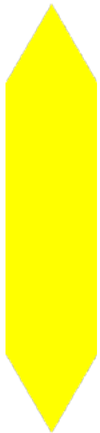
5 6 7 8 9



Before creating a function that produces a seven-segment display for a desired digit, we need to understand how the graphic is composed.

	How could you compose this graphic? What are the various elements it contains? (Help: draw a grid around the graphic.)

Define a `segment` function to create a segment of a certain color (parameter). First, define a function to create the `tip` (an equilateral triangle). Use that `segment` function to compose the final graphic.

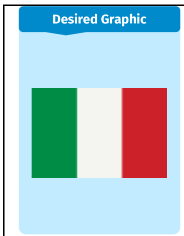


```
def tip


def segment
```

Now we need to think about how to compose the various segments to get the display. With the help of the grid, you will probably have obtained three **columns** (two thin ones on the left and right, a wider one in the middle) and five **rows** (with different heights).

We are able to compose **two** graphics, one beside the other, or one above the other. In case we need to compose **more than two** graphics we can nest (really "make a nest") function calls inside each other, as in the following examples:




```
beside(
  beside(
    green_band,
    white_band
  ),
  red_band
)
```


```
above(
  above(
    black_band,
    red_band
  ),
  golden_band
)
```

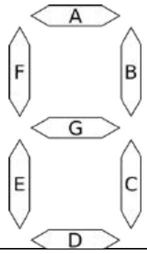
Let us try to define a function named `beside3`, which places three graphics beside each other. **Complete** the body of the following function:

```
def beside3(left, middle, right):
  return beside
```

Define a function named `above5`, which positions five graphics one above each other (using the parameters `g1`, `g2`, `g3`, `g4`, `g5`), with `g1` at the top:

```
def above5(g1, g2, g3, g4, g5):
  return above
```

Choose a digit from zero to nine and **color** the display appropriately:



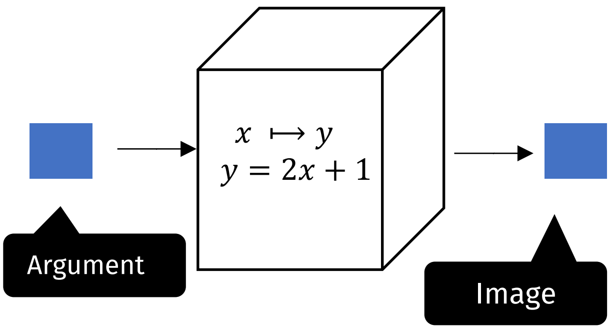
Write the program that creates the seven-segment display that shows that digit. **Use** the `segment` function (with the right color: yellow or black), `beside3` and `above5`.

On the PyTamaro site, go to the “Display a Sette Segmenti” activity, write your program, and test it.

Functions in Mathematics vs Functions in Python

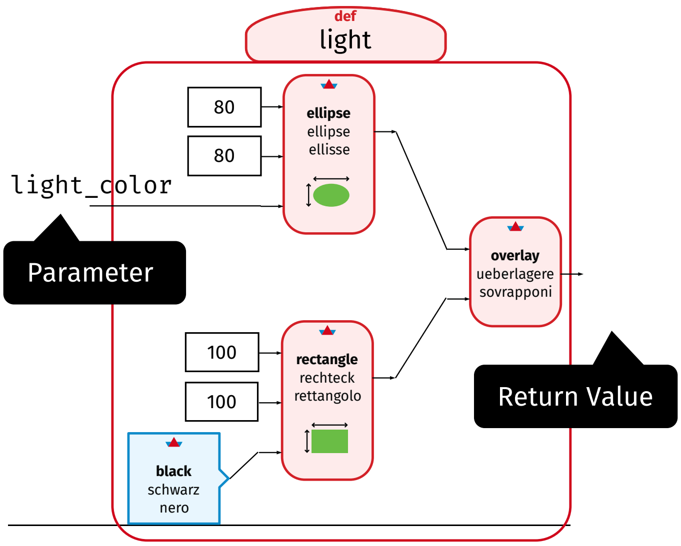
Functions in Mathematics

"The Calculator Box":



A **relation** between two sets A and B, that associates with every element in A one and only one element in B.

Functions in Programming (Python)



Every Value has a Type

We have already used many values, some very different from others.
We can **group** the values into various **sets**.
We can see each set as a **type**.
Below the name of five types, briefly describe what they represent.