Name:	Date:

Automatic Film Generation

Growth Patterns

In Unit 9 we learned how to create lists, by writing down all the values:

List	Element Type	Constant Increment
`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`		
`[True, False]`		
`[6.0, 6.0, 5.0, 5.5, 4.5, 3.5, 4.0, 5.0]`		
`[0, 90, 180, 270]`		
`[0, 30, 60, 90, 120, 150, 180, 210, 240, 270]`		
`[pacman(30), pacman(60), pacman(90), pacman(120)]`		No
`[hsv_color(0, 1, 1), hsv_color(30, 1, 1)]`		
`[rgb_color(0, 0, 0), rgb_color(64, 64, 64)]`	`Color`	No

Some lists contain integers and some are sequences with constant increment.

Compact Description of a List

Describe the characteristics of the following list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Describe the characteristics of the following list: [0, 90, 180, 270]

Describe the characteristic of the following list: [6.0, 6.0, 5.0, 5.5, 4.5, 3.5, 4.0, 5.0]
Is there a clear pattern? What is the main difficulty?

Create a List of Integers Using a Range

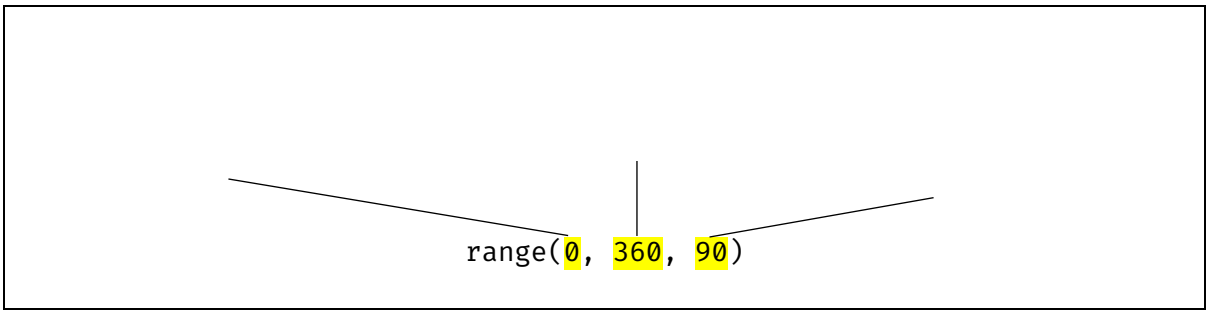
Python has a function with which you can easily create lists of integers, even very long ones.

Link each function call to the list it produces:

`range(0, 10, 1)`
`range(0, 10, 2)`
`range(0, 360, 90)`
`range(1, 11, 1)`
`range(1, 10, 1)`

`[0, 2, 4, 6, 8]`
`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
`[1, 2, 3, 4, 5, 6, 7, 8, 9]`
`[0, 90, 180, 270]`

Describe the meaning of the three parameters of the `range` function:



Similarly in mathematics: in sixth grade we learned to describe a numerical set with the following notation.

$$A = \{1; 2; 3; 4; 5; 6; 7; 8; 9; 10\} = \{x \in N | 0 < x < 11\}$$

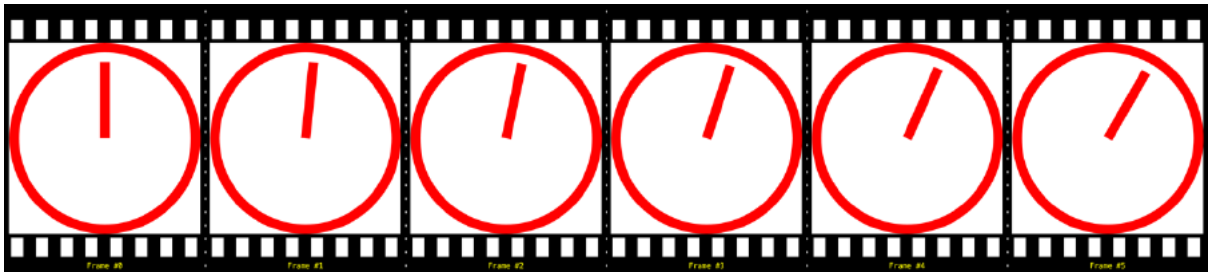
$$B = \{0; 90; 180; 270\} = \{x \in N |\ x \text{ è } multiplo\ di\ 90\ e\ x < 360\}$$

Call the range function to produce the following lists:

[2, 4, 6, 8, 10]	
[0, 3, 6, 9]	
[0, 30, 60, 90, 120, 150, 180, 210, 240, 270]	
[1, 3, 5, 7, 9]	

Use the range Function to Create a Film!

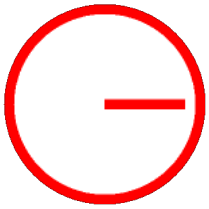
Let's try to create a film of a clock with a rotating hand. Here are the first six frames:



How many frames does it take to show a full rotation if the hand jumps forward one second in each frame?

A movie is a list of **graphics**. Assume there exists a function, `clock`, that can produce a graphic of a clock with the hand positioned at the required second.

Complete the table:

`clock(0)`		`clock(15)`	`clock(50)`
			

With the `clock` function we can create an animation (film) in this way:

```
show_animation(
  [ clock(0), clock(1), clock(2), clock(3), clock(4), ... ]
)
```

With this method we have to do a lot of work and list so many calls to the clock function. How boring...

With `range` we can easily create a list containing a list of numbers. Can we somehow use it to easily create a list of graphics?

Of course! We are programming, **we can do almost anything**!

We can proceed in two steps:

1. Use `range` to create a list of integers
2. Turn the list of numbers into graphics, calling `clock` for each element in the range

Step 1: Create a list of numbers

Call `range` to produce the necessary list of integer numbers:

Step 2: Map from the list of numbers to the list of clock graphics

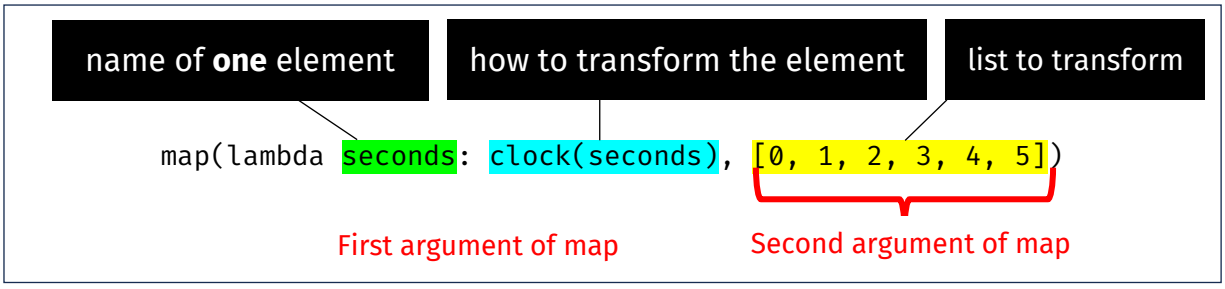
Now you have a list with the desired numbers: [0, 1, 2, 3, 4, 5, ..., 58, 59].

Python has a fantastic feature that **transforms** a list of one type (for example, a list of numbers) into another type of list (for example, a list of graphics). **Each element is transformed individually**.
This transformation is also called **map** (not in the sense of a geographic map, but a mathematical function).

How can we transform ("map") an integer number into a clock?
For example, how can we transform the integer number 45 into a clock graphic with the hand indicating 45 seconds? Write the code:

We simply call our `clock` function. This function takes an integer number as an argument and produces as its output value a graphic showing the clock!

Now we need to call the `clock` function for each item in our list.
Here's how it works:



We are using the `map` function that "maps," that is, transforms, a list of elements into another list. To call `map` we need two arguments.

The **second argument** of map (highlighted in yellow) is the list to be transformed.
The **first argument**, on the other hand, tells how to transform **a single** list element.
It is a small, unnamed function that we write directly there.
Instead of `def <name>`, we start with the `lambda` keyword.
We then have to choose the name for the parameter: e.g., `seconds`.
We put the colon `:` and then directly the body of the function (light blue), without `return`. We need to specify how to transform a single element, using the parameter (in green) as an argument to the `clock` function (light blue).

Now combine **map with lambda** and the call of the `range` function to produce the list of clock graphics from 0 to 59 seconds:

```
```

In the space where you specify the list to transform, you can use `range`.
Now use the list produced by `map` as an argument to call `show_animation`, and you will see your animated clock!

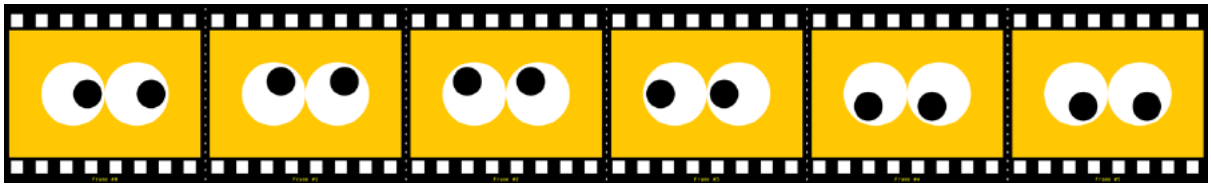
Now try it on the PyTamaro web site ("Animazione Orologio" activity).

Small but important clarification: for reasons of execution speed, the `range` and `map` functions return values that are "almost" lists, but not quite.
Whenever you really need a list (as with `show_animation`), use the `list(...)` function to get a real list. For example:

```
show_animation(list(map(..., ...)))
```

Animated Rolling Eyes

Let's try to animate the following short film with 6 frames:



Use `map` and `range` to create a list of angles and the function `framed_eyes(angle_rotation)` that you programmed in Unit 9.

```
show_animation(
  list(map(
    lambda
```

This little film contains only 6 frames. Instead of creating a rolling eyes film of 6 frames, try 36 frames (how does the `range` call change?).

```
show_animation(
  list(map(
    lambda
```

By changing only one topic of the `range` call it becomes super easy to create smooth animated movies! Try it on the PyTamaro web site!

Animated Pacman!

You have already created a short Pac-Man animation. Now try creating one where the mouth gradually opens **and then** closes. You need to create a list of the mouth angles.

- For the first half, the angles of the mouth should gradually increase (mouth **opening**).
- For the second half, the angles of the mouth should gradually decrease (mouth **closing**).

With the addition symbol `+` you can concatenate lists. For example:

```
my_preferred_numbers   = [1, 2, 3]
your_preferred_numbers = [9, 4, 2]
our_preferred_numbers  = my_preferred_numbers + your_preferred_numbers
```

The elements of `our_preferred_numbers` are [1, 2, 3, 9, 4, 2].

How can we create a list with decreasing numbers without listing them all? Can the range function do this?

Connect the range calls to the following lists. Warning: two of the calls produce the same list!

`range(5, 0, -1)`
`range(1, 0, -1)`
`range(1, 0, -2)`
`range(10, -1, -1)`
`range(10, -1, -2)`

`[5, 4, 3, 2, 1]`
`[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]`
`[10, 8, 6, 4, 2, 0]`
`[1]`

Use `range` to produce the angles of a mouth that is closing: [120, 90, 60, 30]

```
```

Write a function that produces a list of integers representing the angles of Pac-Man's mouth, such as [0, 30, 60, 90, 120, 90, 60, 30].
Remember that `range` produces an “almost list”: to get a proper list, use `list(...)`.

```
def mouth_angles_steps_of_30():

  return list(range(    ,    ,    )) + list(range(    ,    ,    ))
```

Write a more general function that specifies how long a step is:

```
def mouth_angles(step):

  return list(range(    ,    , step)) + list(range(    ,    , -step))
```

Using your `mouth_angle` and `pacman` functions, developed in the previous booklets, create an animation with a step size of 10 degrees.

```
show_animation(
  list(map(
    lambda
```

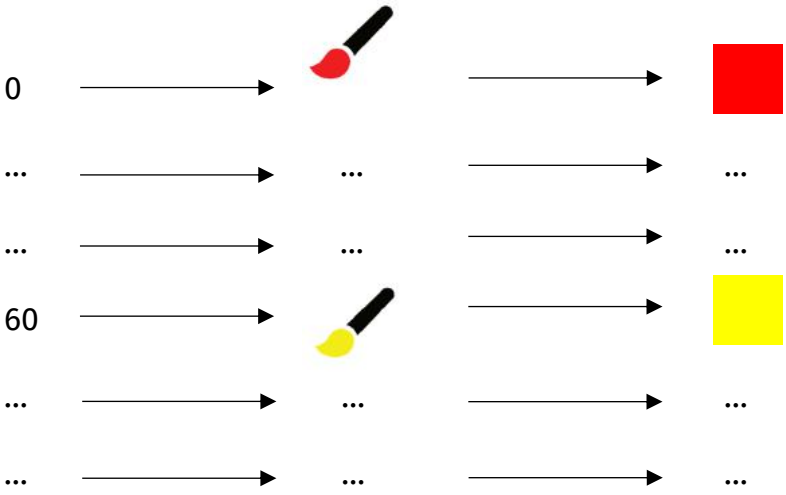
Now try this out on the PyTamaro web site.

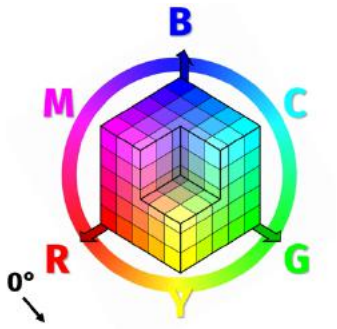
Colored Squares

Let's now try to create an animation that shows all the colors of the rainbow!
Do you remember the HSV color model?

Follow these steps:

1. **Create** a list of numbers from 0 to 359 inclusive, which will serve as angles
2. **Map** each number, which indicates hue, to the corresponding color. Use 1.0 for both saturation and value.
3. **Map** each color to a square graphic of that color





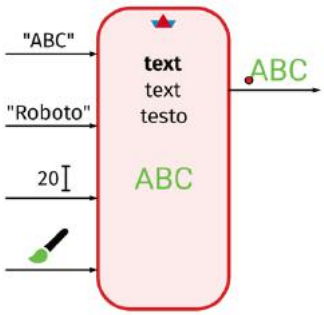

```
angles =



colors =



graphics =









show_animation(list(graphics))
```

Now try this out on the PyTamaro web site.

Animated Spelling

Create an animated version of spelling your name!
Each frame will display a graphic containing a letter.

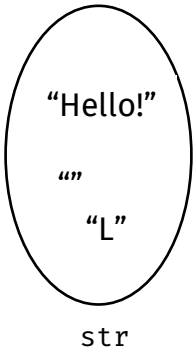
To create a graphic with text, PyTamaro offers the `text` function:



Call the function to produce a graphic showing “Hello!” in black, written in font “Fira Sans” at size 32.

```
```

Every textual part should be enclosed in **quotation marks**.
Such textual values belong to a new type called `str`
(string, or sequence of characters,
which can be letters, numbers, spaces…).



Create a list where each element is a string containing only one letter of your name.
Then, **map** this list of strings into a list of graphics. To keep the same size across frames, create a frame on which to overlay the letters.
Finally, show it as an animation (possibly slowing it down a bit)!

```
letters =


graphics =



show_animation(list(graphics))
```

Now try this out on the PyTamaro web site.