CMPU 101 § 53 · Computer Science I

Expressions, Values, and Names

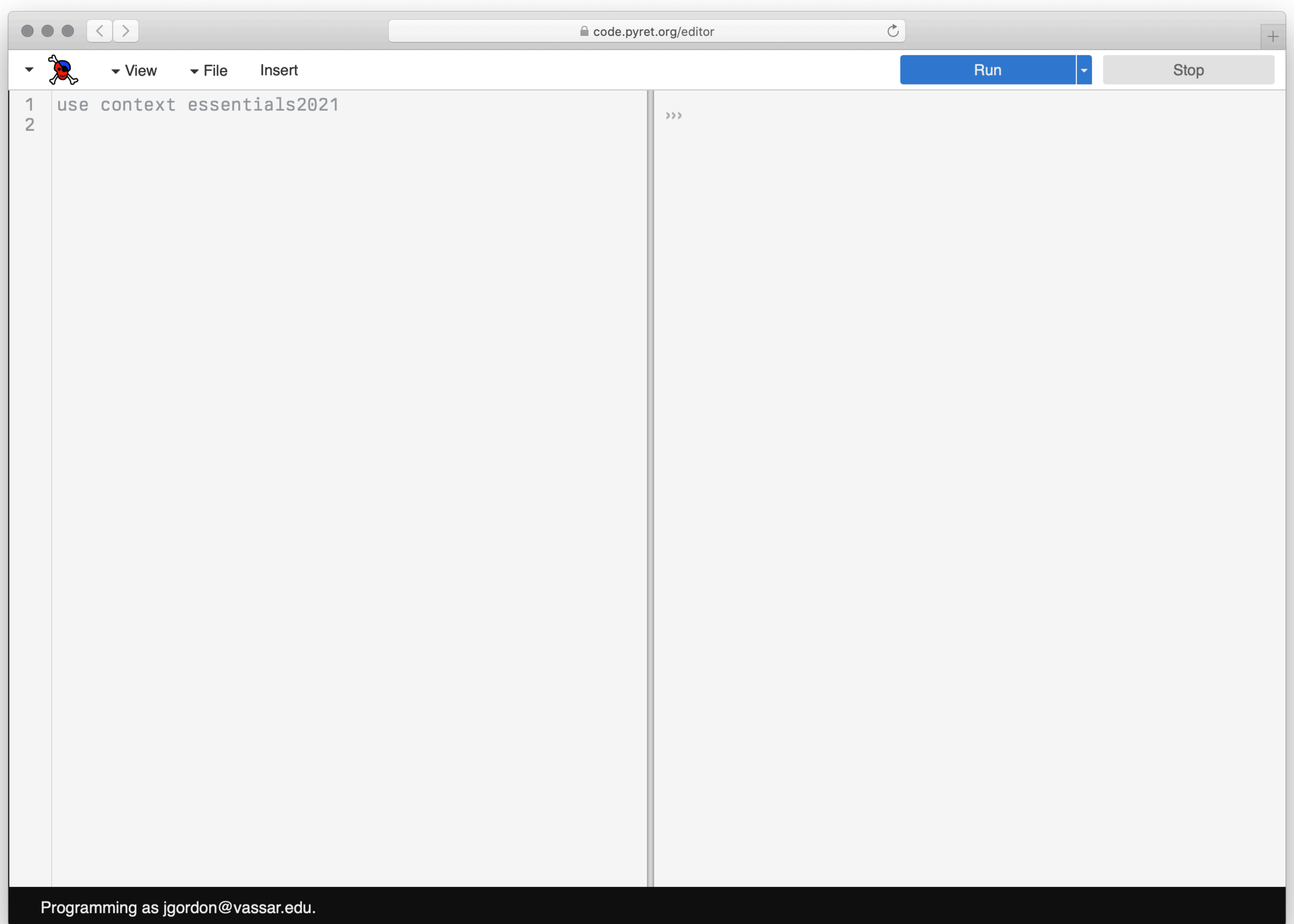
23 January 2024

Where are we?

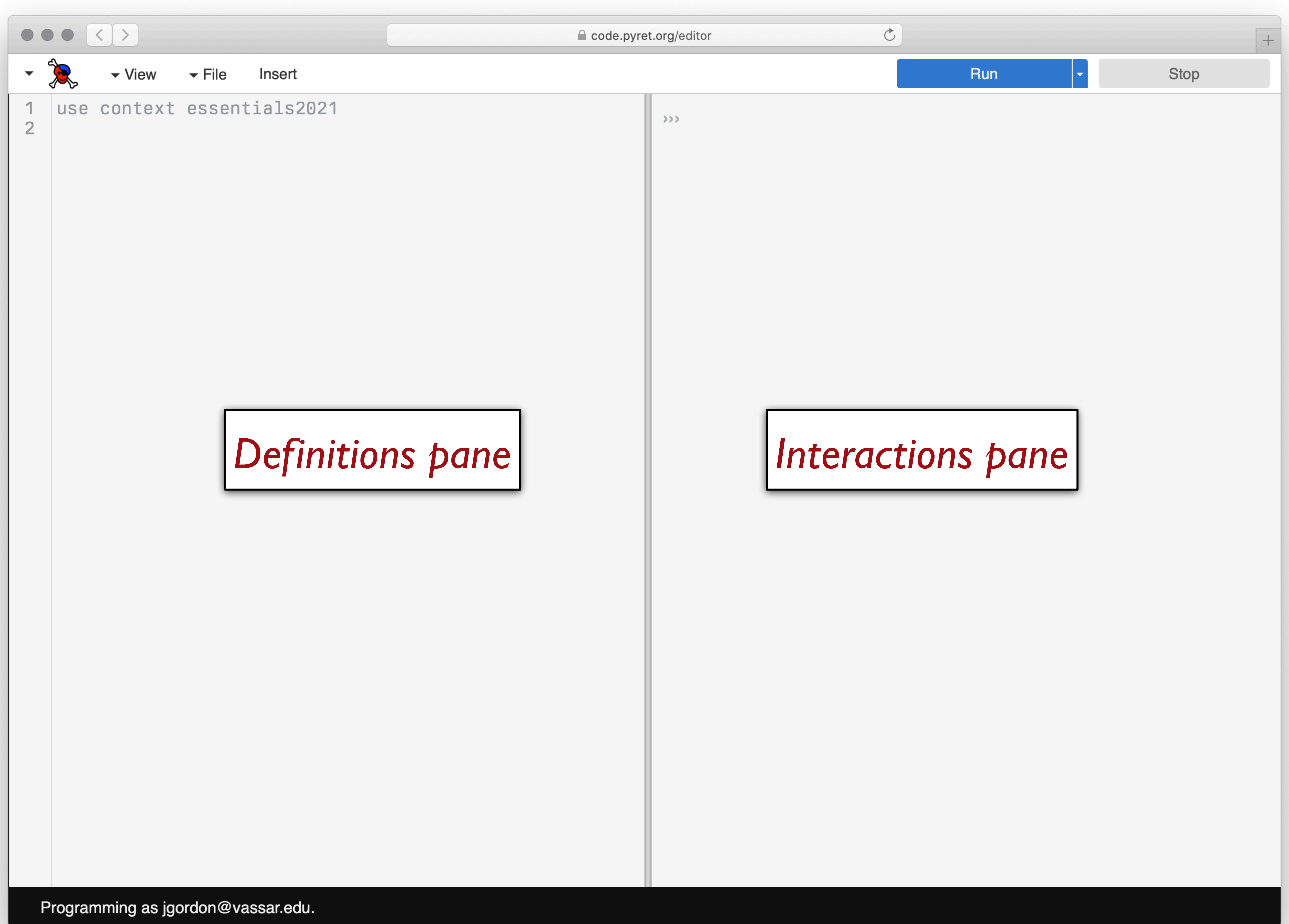
A *program* instructs a computer to do something.

- For the computer to carry out these instructions, they need to be very specific.
- But programs also need to be understood by people, so they need to be readable!

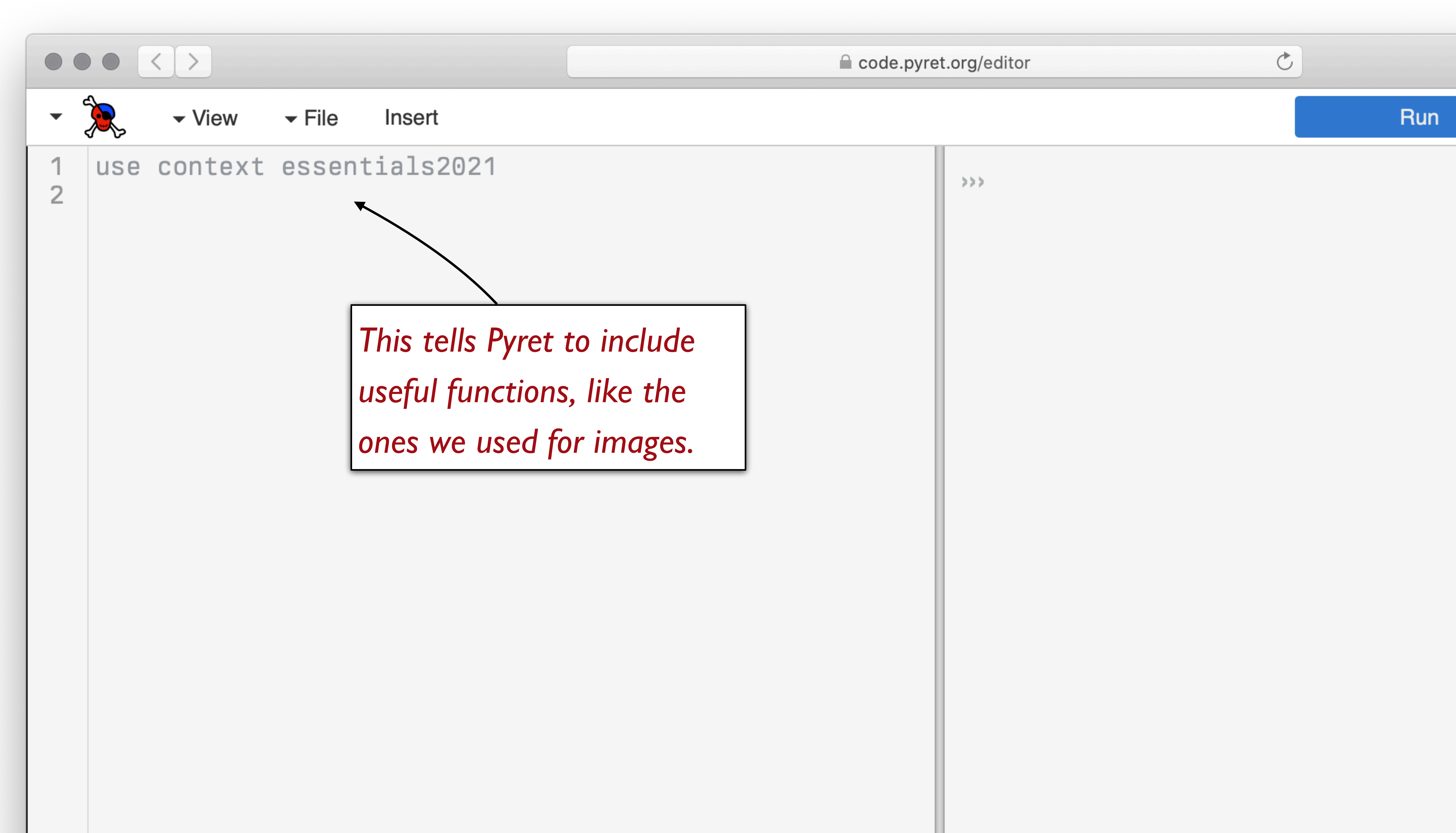
We write a program in a *programming language* and
we run it in a *programming environment*.

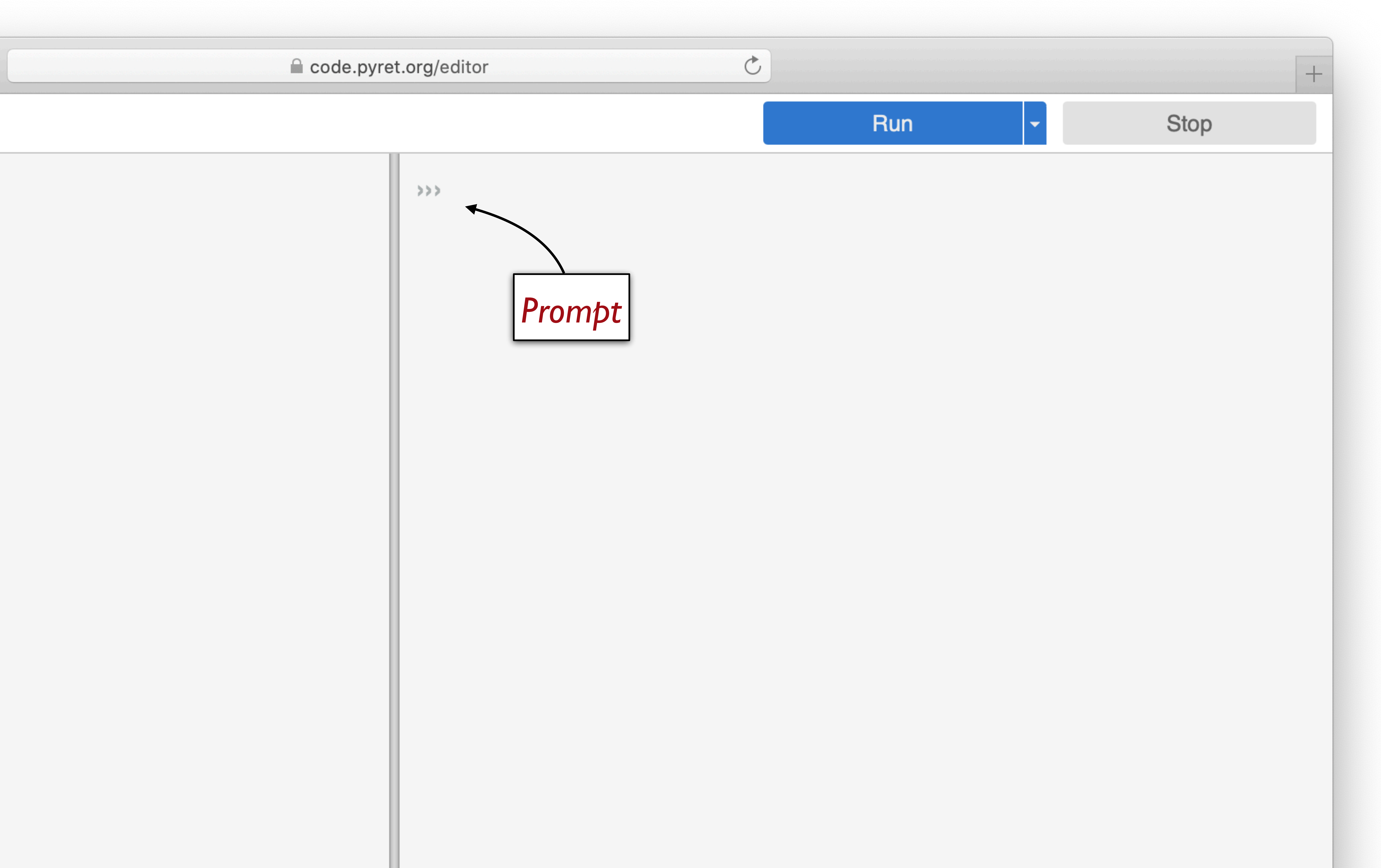


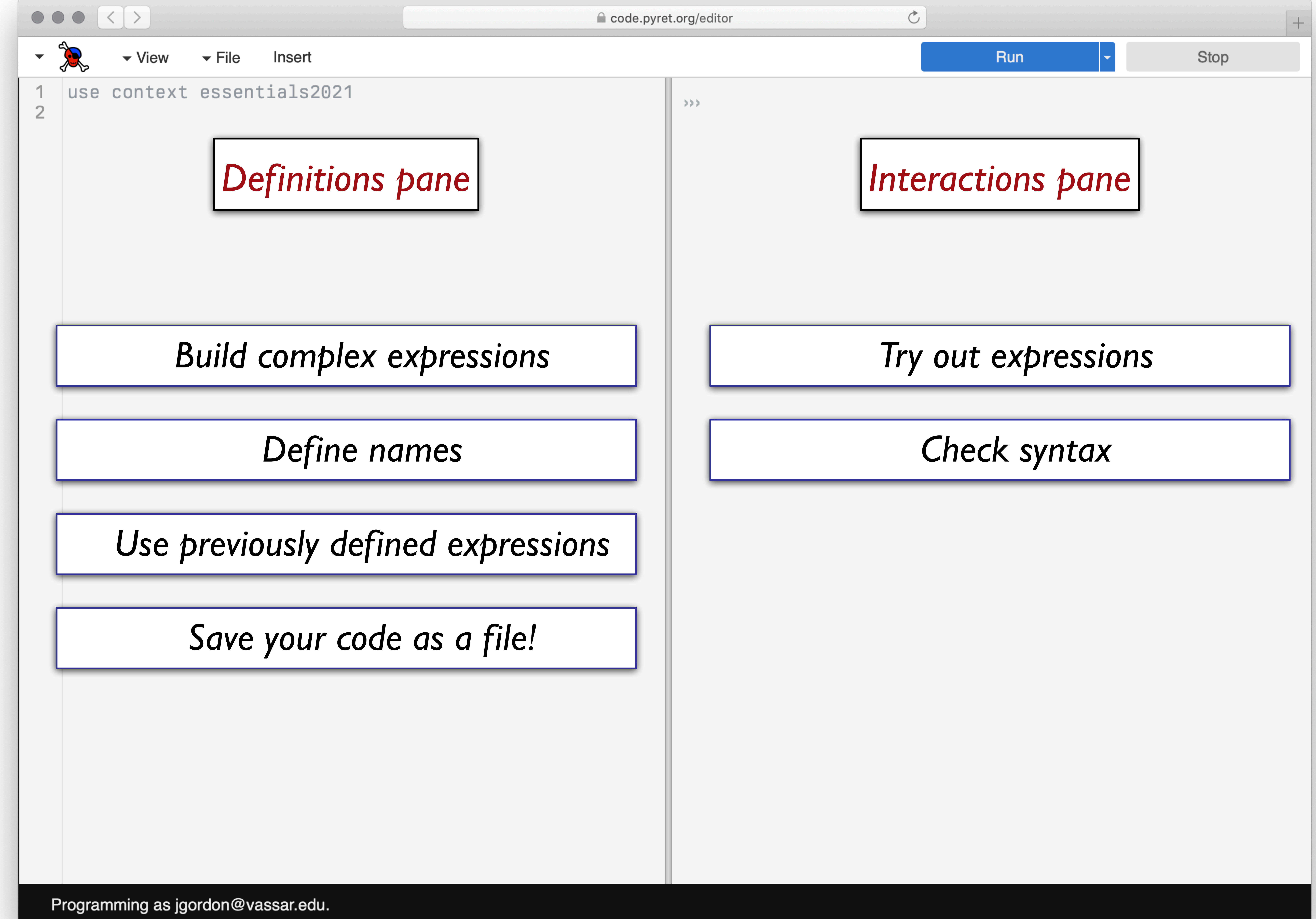
code.pyret.org



code.pyret.org


code.pyret.org/editor
View
File
Insert
Run
use context essentials2021
This tells Pyret to include useful functions, like the ones we used for images.


code.pyret.org/editor
Run
Stop
›››
Prompt


code.pyret.org/editor
View
File
Insert
Run
Stop
1 use context essentials2021
2
Definitions pane
Interactions pane
Build complex expressions
Try out expressions
Define names
Check syntax
Use previously defined expressions
Save your code as a file!
Programming as jgordon@vassar.edu.

Which pane would I use if…

Which pane would I use if…

I want to see if I can make a blue circle?

Which pane would I use if…

I want to see if I can make a blue circle?

I want to define `my-shape` as a blue circle and use it later in my code?

Which pane would I use if…

I want to see if I can make a blue circle?

I want to define `my-shape` as a blue circle and use it later in my code?

I want to see if Pyret will accept this: `print "5"`?

Which pane would I use if…

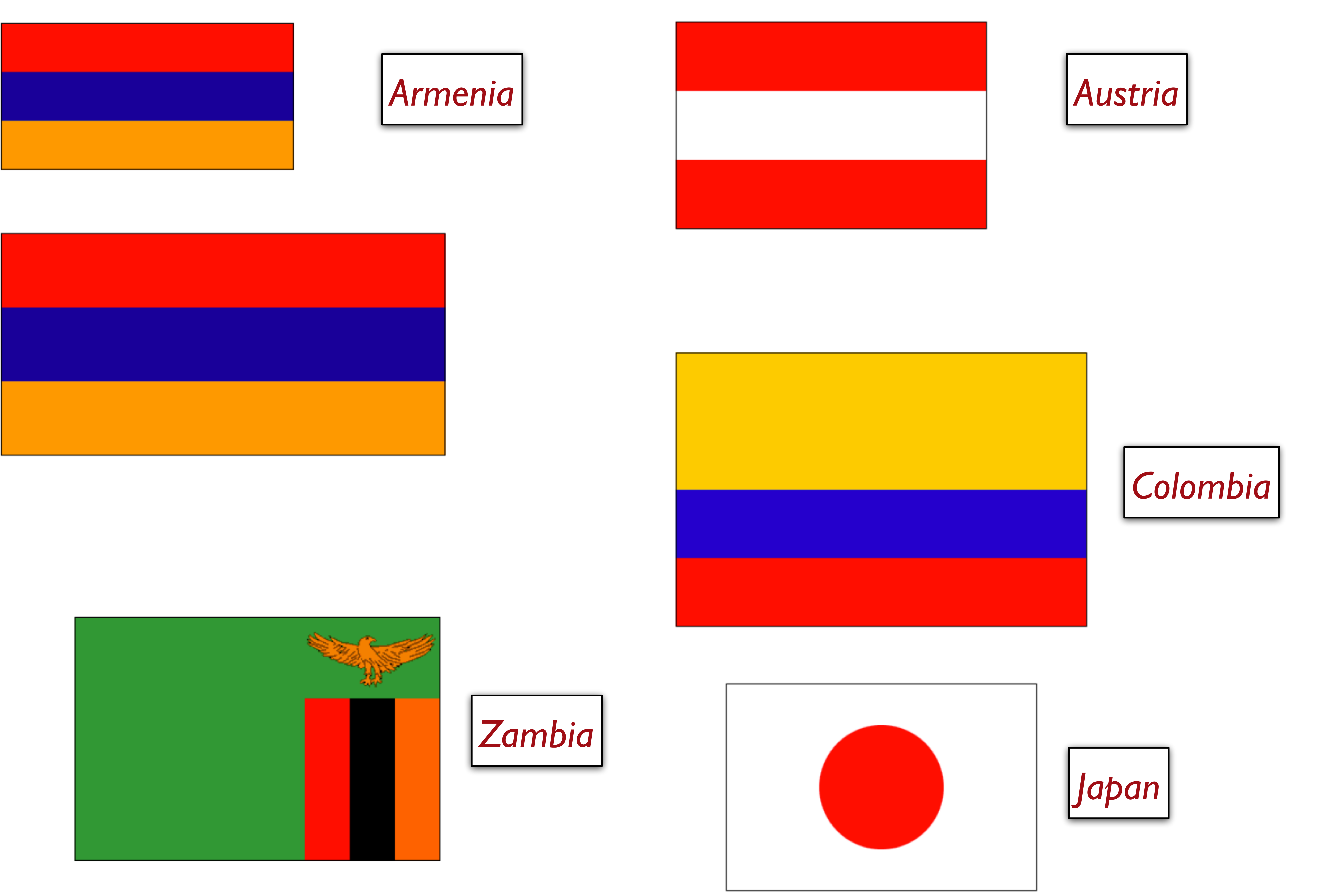
I want to see if I can make a blue circle?

I want to define `my-shape` as a blue circle and use it later in my code?

I want to see if Pyret will accept this: `print "5"`?

I want to start my assignment now and finish it later?

Starting to program


Armenia
Austria
Colombia
Zambia
Japan

We're trying to make sense of the problem.

We start with the *data* before we dive in to try to *do* it.

We might want to compute the heights of the stripes from the overall flag dimensions, which means we need to write programs over ***numbers***.

We need a way to describe ***colors*** to our program.

We need a way to create images based on simple ***shapes*** of different colors.

We might want to compute the heights of the stripes from the overall flag dimensions, which means we need to write programs over *numbers*.

We need a way to describe *colors* to our program.

We need a way to create images based on simple *shapes* of different colors.

An individual number like **5** is a *value* – it can't be computed any further.

`(3 + 4) * (5 + 1)` is an *expression* – a computation that produces an answer.

A program just consists of one or more computations you want to run.

```
>>> 3 + 4 * 5 + 1
```

Reading this expression errored:

```
interactions://1:0:0-0:13
1  3 + 4 * 5 + 1
```

The * and + operations are at the same grouping level. Add parentheses to group the operations, and make the order of operations clear.

```
>>> (3 + 4) * (5 + 1)
42
```

```
››› num-min(5, 9)
5
```

We might want to compute the heights of the stripes from the overall flag dimensions, which means we need to write programs over *numbers*.

We need a way to describe *colors* to our program.

We need a way to create images based on simple *shapes* of different colors.

Names can be given as ***text strings***, e.g., `"blue"`.

We might want to compute the heights of the stripes from the overall flag dimensions, which means we need to write programs over *numbers*.

We need a way to describe *colors* to our program.

We need a way to create images based on simple *shapes* of different colors.

```
››› include image
›››
```

You only need to type this if you haven't pressed "Run" to get the image functions from the context line in the definitions pane.

```
››› include image
›››
```

You only need to type this if you haven't pressed "Run" to get the image functions from the context line in the definitions pane.

```
››› include image

››› circle(50, "solid", "red")
```

You only need to type this if you haven't pressed "Run" to get the image functions from the context line in the definitions pane.

```
››› include image
››› circle(50, "solid", "red")
```

We can manipulate images much like we can manipulate numbers.

- Numbers can be added, subtracted, etc.
- Images can overlaid, rotated, flipped, etc.

Evaluation

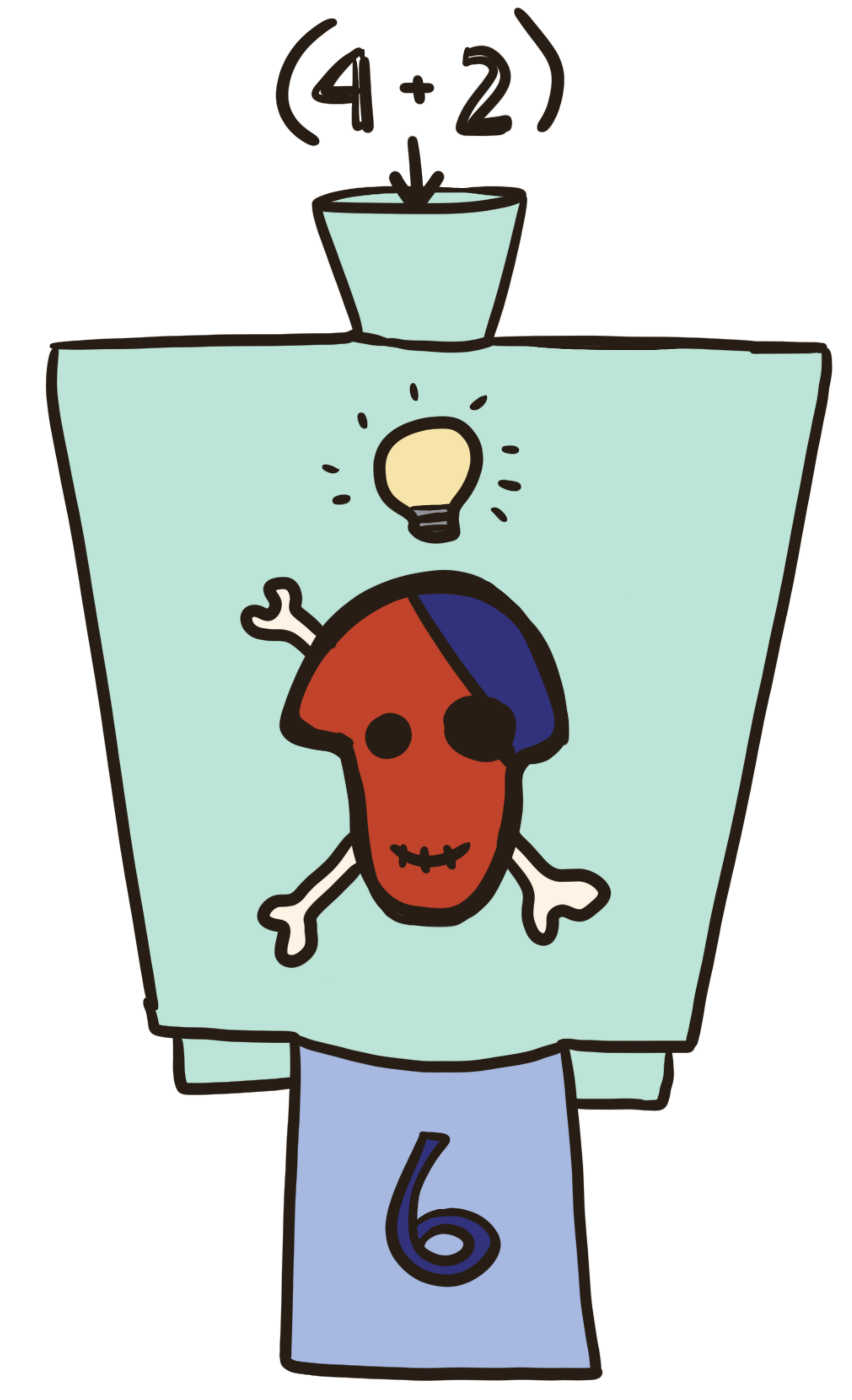
How does something like `(4 + 2) / 3` work?

What is the operator / dividing?

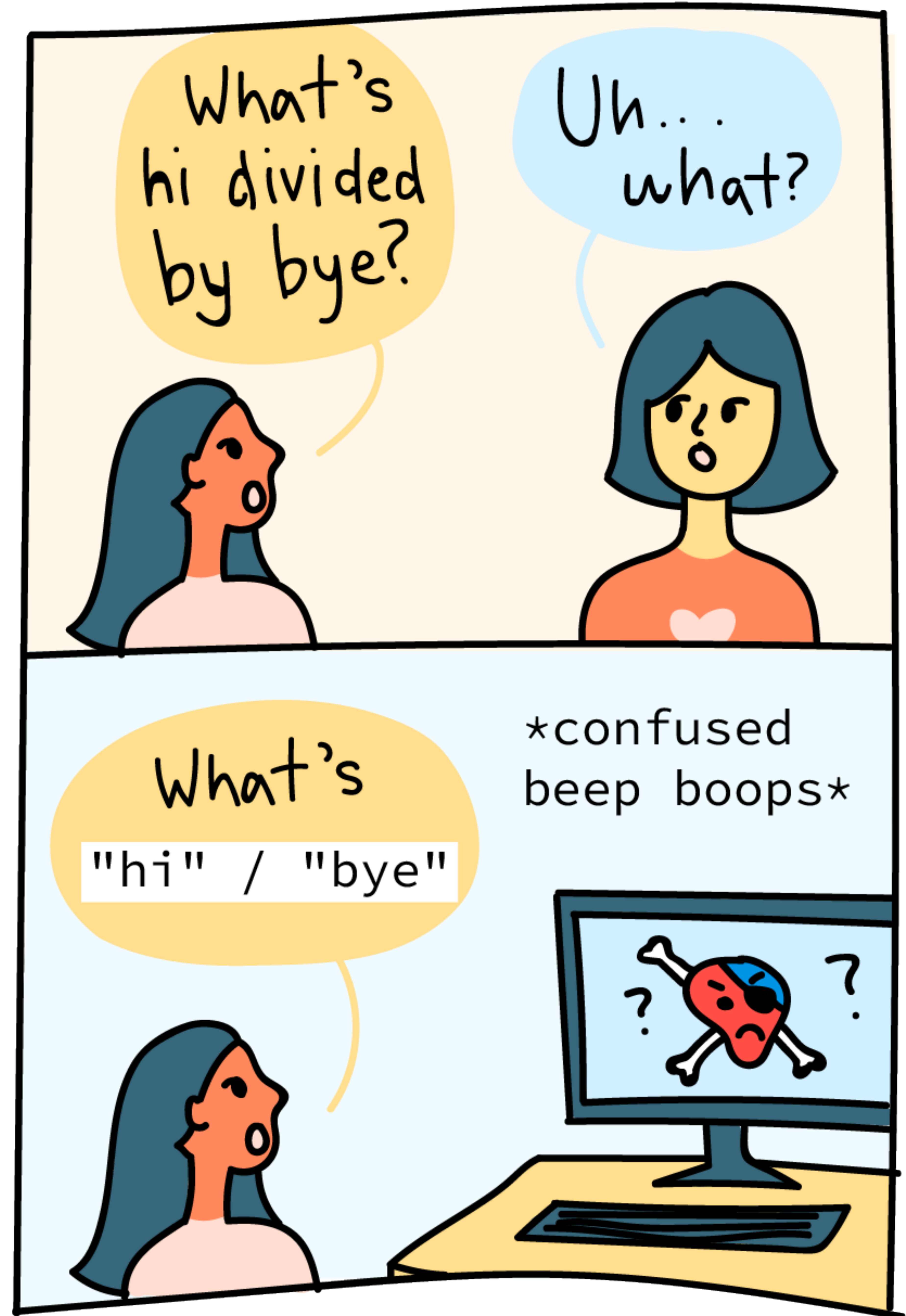
Shouldn't / expect two numbers?

Even though `(4 + 2)` isn't a number, it's an expression that *evaluates* to a number.

This works for all data types, not just numbers!



Operations may only work on certain types of data!



When we write complex expressions, Pyret evaluates them from the inside out:

```
7 + (6 / (1 + 1))
```

When we write complex expressions, Pyret evaluates them from the inside out:

```
  7 + (6 / (1 + 1))
→ 7 + (6 / 2)
```

When we write complex expressions, Pyret evaluates them from the inside out:

```
  7 + (6 / (1 + 1))
→ 7 + (6 / 2)
→ 7 + 3
```

When we write complex expressions, Pyret evaluates them from the inside out:

```
  7 + (6 / (1 + 1))
→ 7 + (6 / 2)
→ 7 + 3
→ 10
```

When we write complex expressions, Pyret evaluates them from the inside out:

```
  beside(
    circle(10, "solid", "red"),
    circle(10, "solid", "blue"))

→ beside(    ,
    circle(10, "solid", "blue"))

→ beside(    ,    )

→
```

What’s in a name?

```
>>> x = 5
```

```
>>> x = 5
>>>
```

There's no output from entering a definition.

```
››› x = 5
›››
```

Directory

Name	Value
x	5

There's no output from entering a definition.

It only has the side effect of telling Pyret to associate the name with the value in the program directory.

```
>>> x = 5
>>> x
```

Directory

Name	**Value**
x	5

```
››› x = 5
››› x
5
```

When you use the name later, Pyret looks it up in the directory and substitutes the value it finds.

Directory

Name	***Value***
x	5

› › ›

Directory

Name	***Value***

```
>>> fname = "Grace"
>>>
```

Directory

Name	Value
fname	"Grace"

```
>>> fname = "Grace"
>>> lname = "Hopper"
>>>
```

Directory

Name	***Value***
fname	"Grace"
lname	"Hopper"

```
>>> fname = "Grace"
>>> lname = "Hopper"
>>> fname + " " + lname
```

Directory

Name	**Value**
fname	"Grace"
lname	"Hopper"

```
>>> fname = "Grace"
>>> lname = "Hopper"
>>> fname + " " + lname
→   "Grace" + " " + lname
```

Directory

Name	**Value**
fname	"Grace"
lname	"Hopper"

```
>>> fname = "Grace"
>>> lname = "Hopper"
>>> fname + " " + lname
→   "Grace" + " " + lname
→   "Grace " + lname
```

Directory

Name	**Value**
fname	"Grace"
lname	"Hopper"

```
>>> fname = "Grace"
>>> lname = "Hopper"
>>> fname + " " + lname
→   "Grace" + " " + lname
→   "Grace " + lname
→   "Grace " + "Hopper"
```

Directory

Name	***Value***
fname	"Grace"
lname	"Hopper"

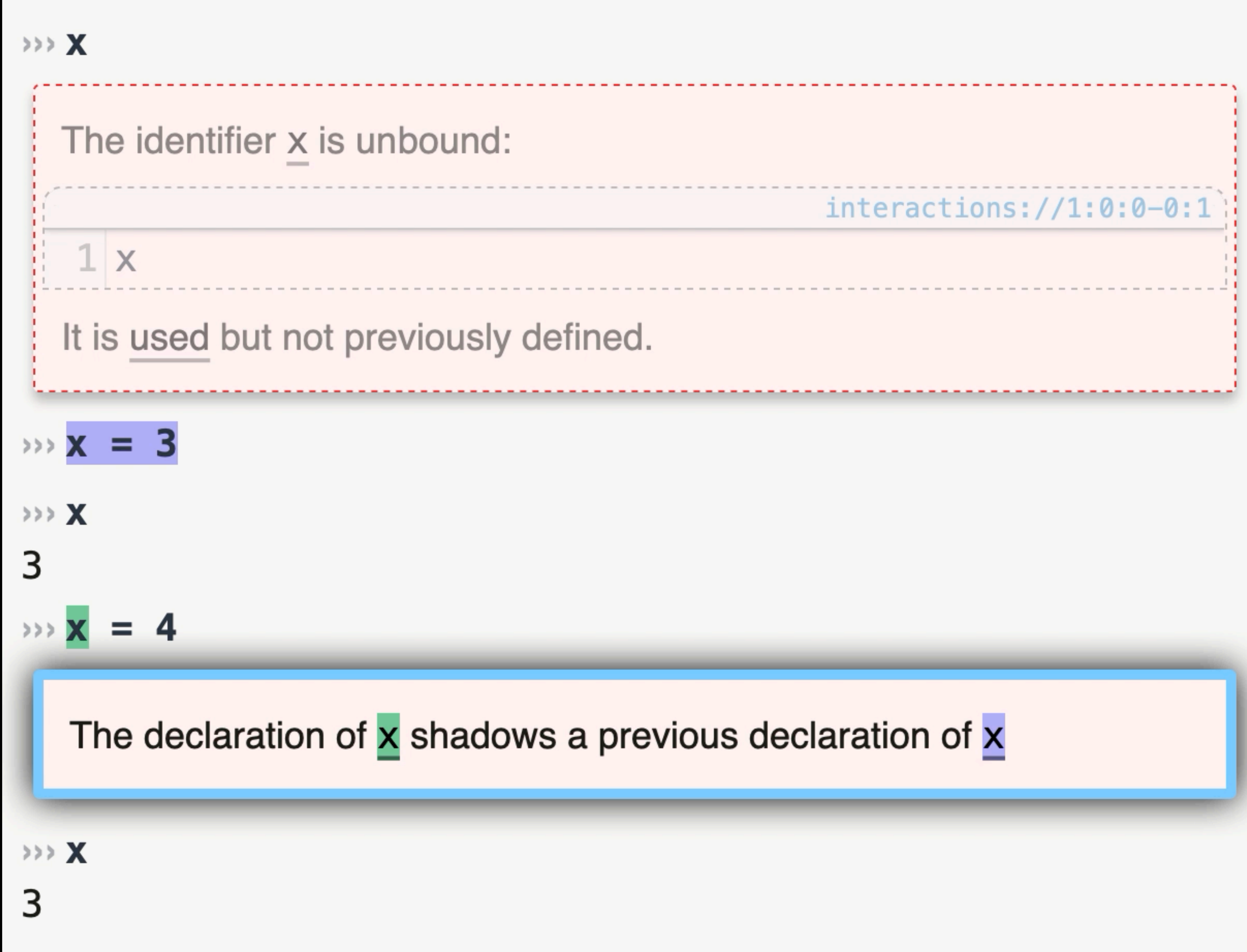
```
>>> fname = "Grace"
>>> lname = "Hopper"
>>> fname + " " + lname
→   "Grace" + " " + lname
→   "Grace " + lname
→   "Grace " + "Hopper"
→   "Grace Hopper"
```

Directory

Name	Value
fname	"Grace"
lname	"Hopper"

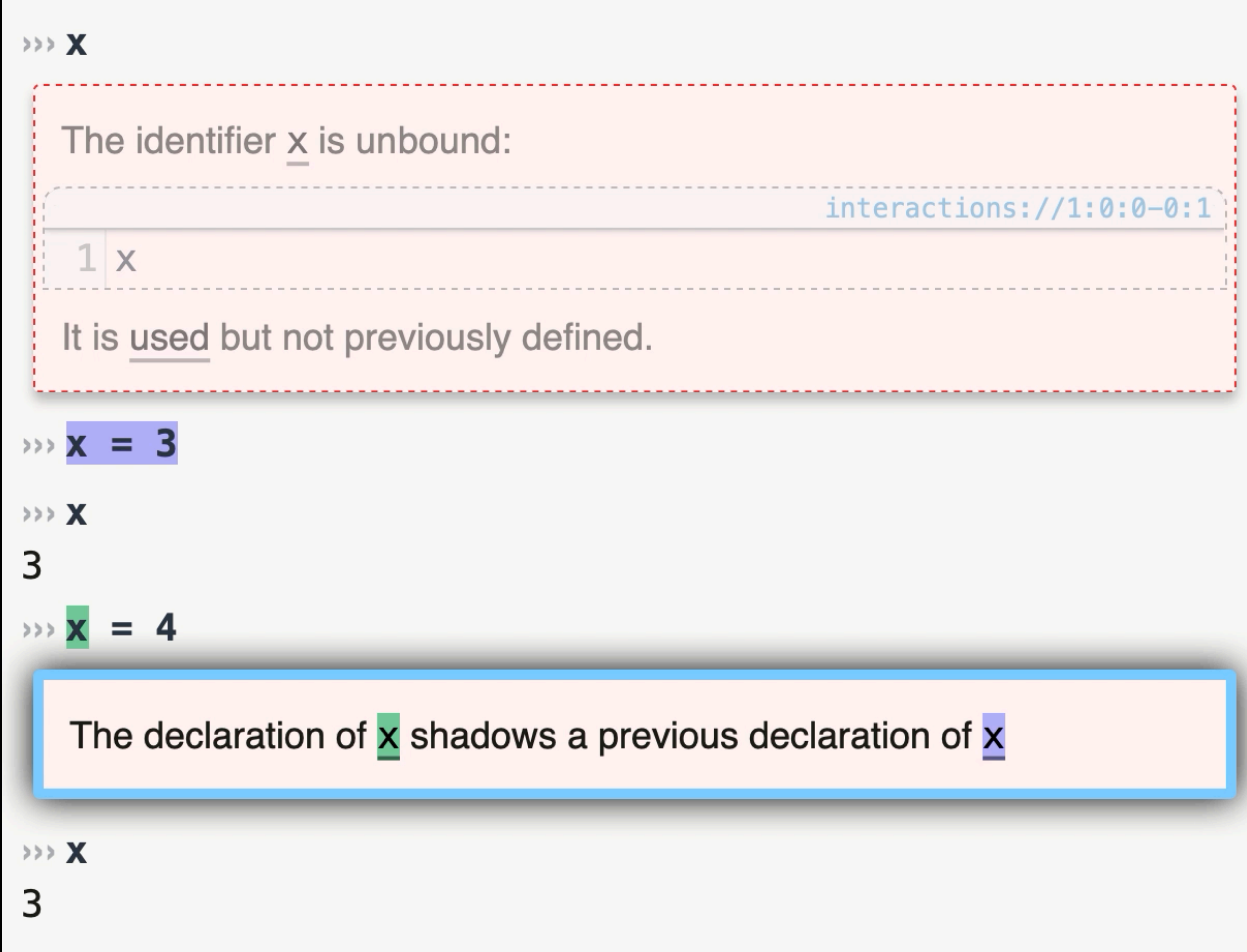
Names must be given a value before being used.

In Pyret, names are *immutable*, which means they can only be defined once.



Names must be given a value before being used.

In Pyret, names are *immutable*, which means they can only be defined once.



Names are arbitrary

The following is silly, but legal:

```
>>> five = 6
>>> five
6
>>> six = 5
>>> six
5
```

Several constants may have the same value:

```
››› seven = 7
››› seven
7
››› septem = 7
››› septem
7
```

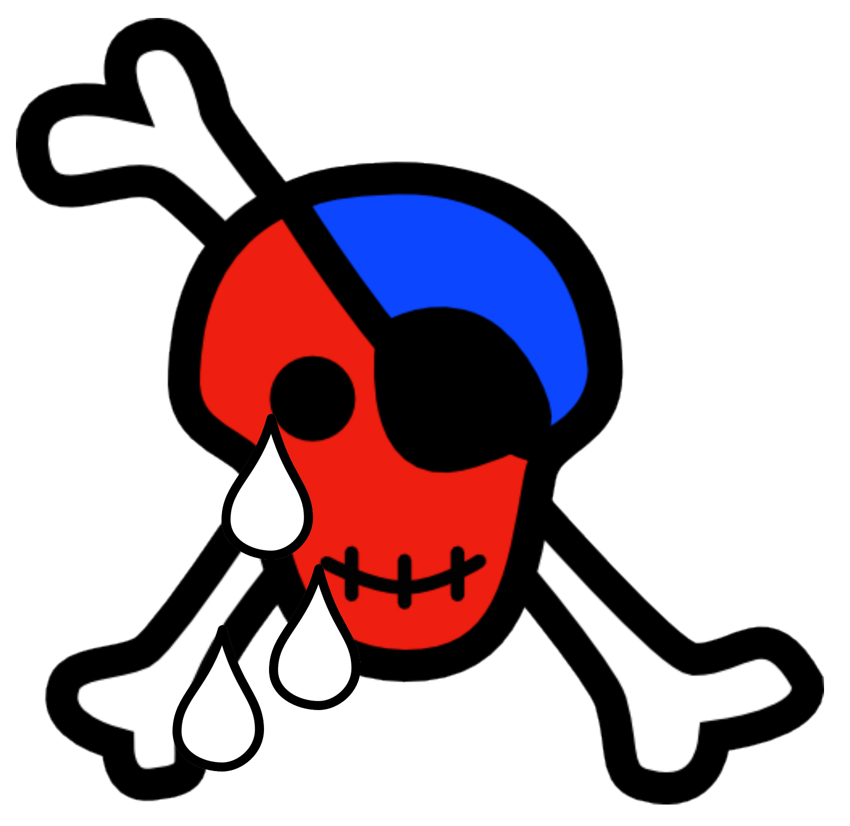
Every programming language has its own conventions for names.

In Pyret, names are lowercase with words joined by hyphens, e.g.,

```
this-is-a-good-name

this_makes_bonny_cry

thisIsACrimeAgainstPyret
```



Concept check

We can define the names

```
width = 400
height = 600
```

Now if we write

```
width * height
```

it gets evaluated:

```
→ 400   * height
→ 400   * 600
→ 240000
```

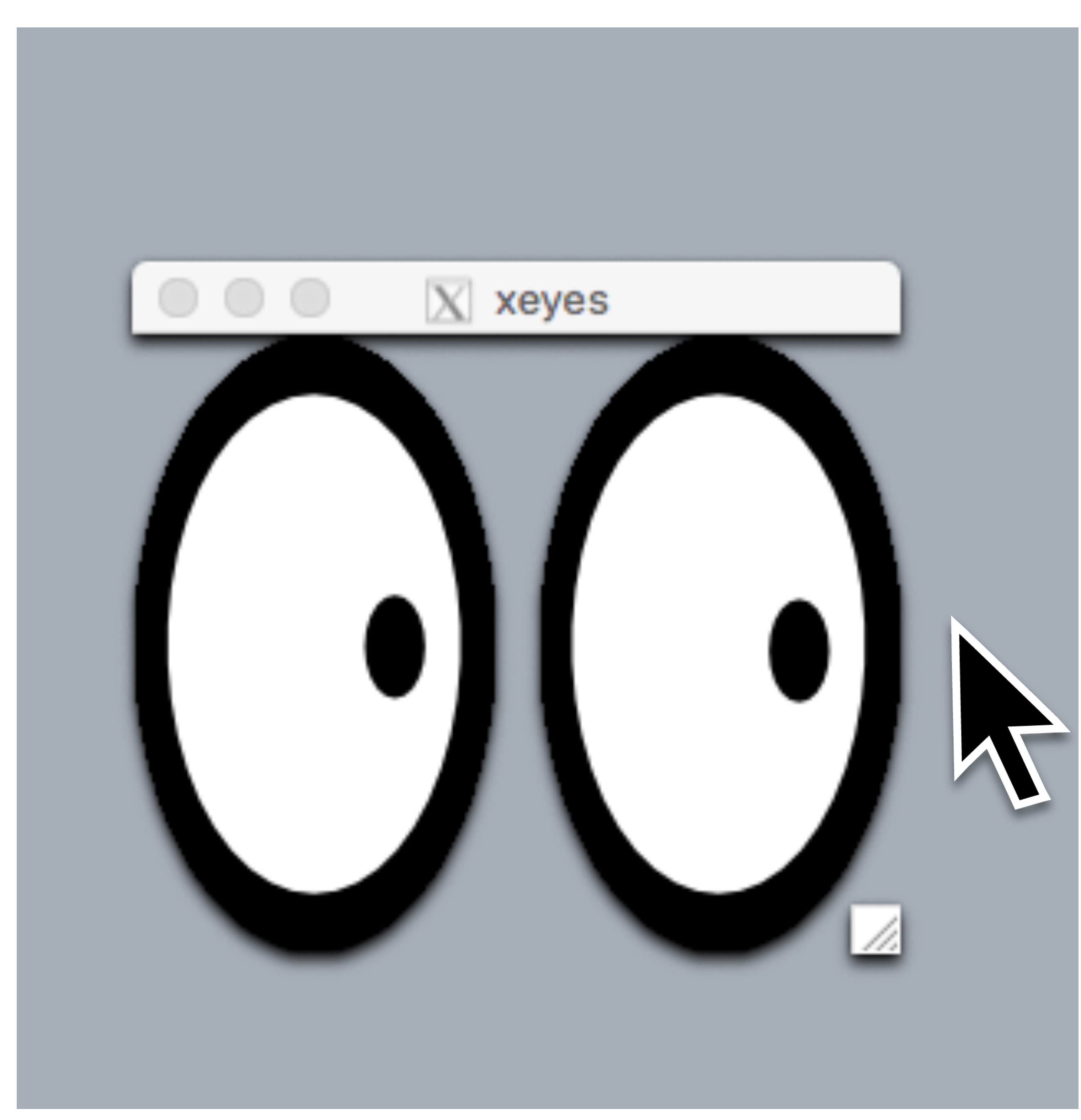
What if we use another name?

```
width = 400
height = 600
area = width * height
```

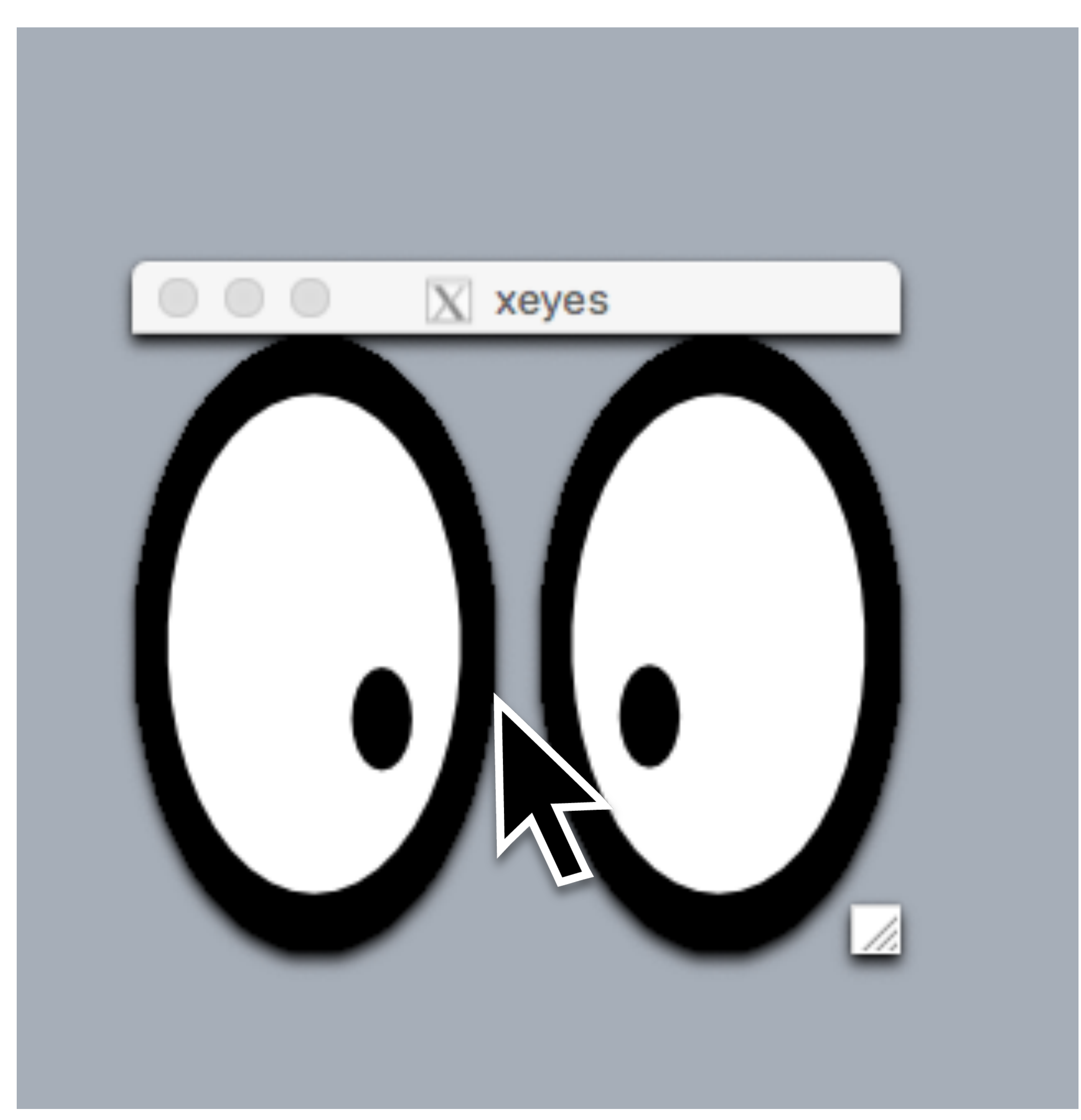
Does Pyret associate the name **`area`** with the expression **`width * height`** or with the number **`240000`**?

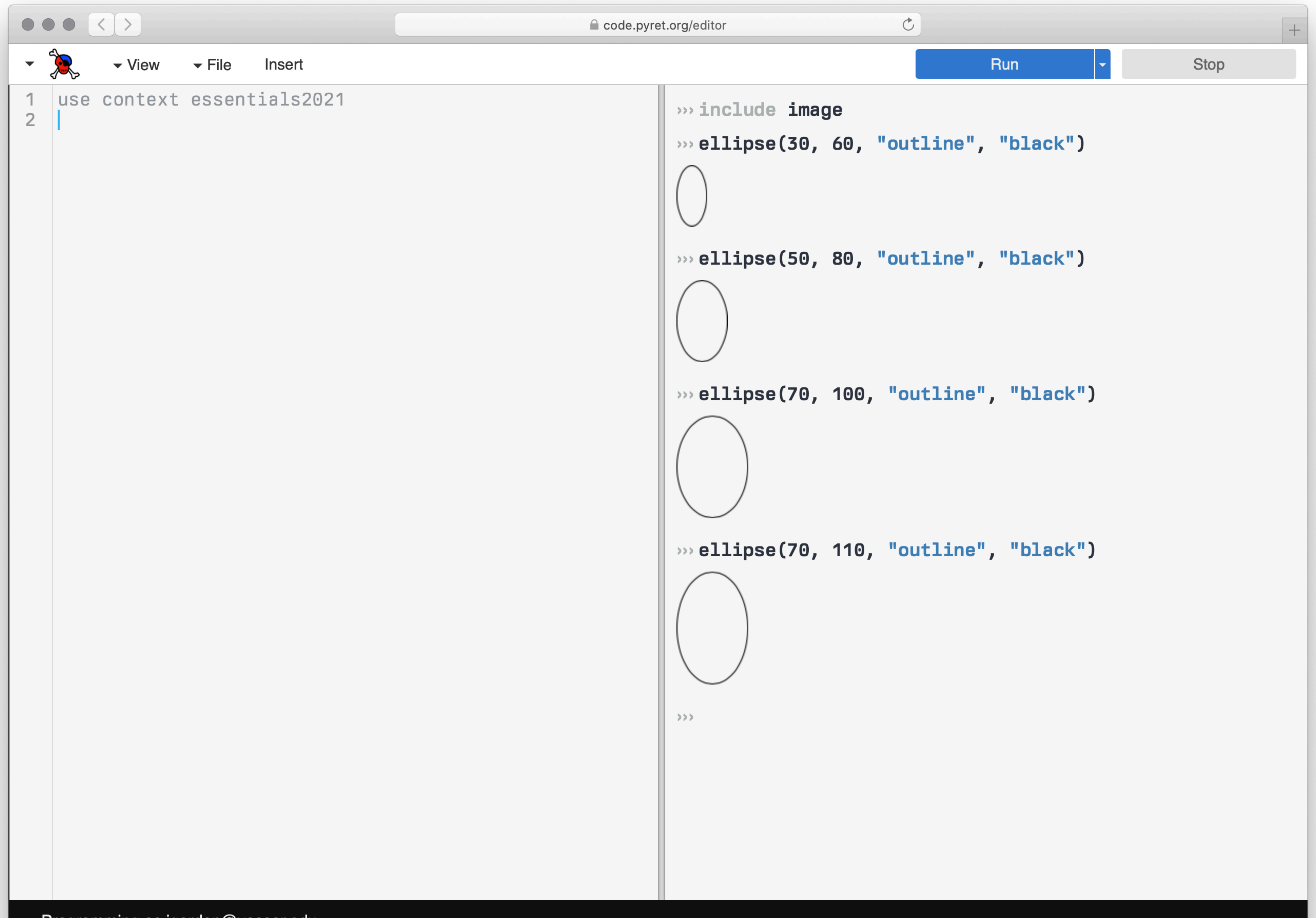
Exercise

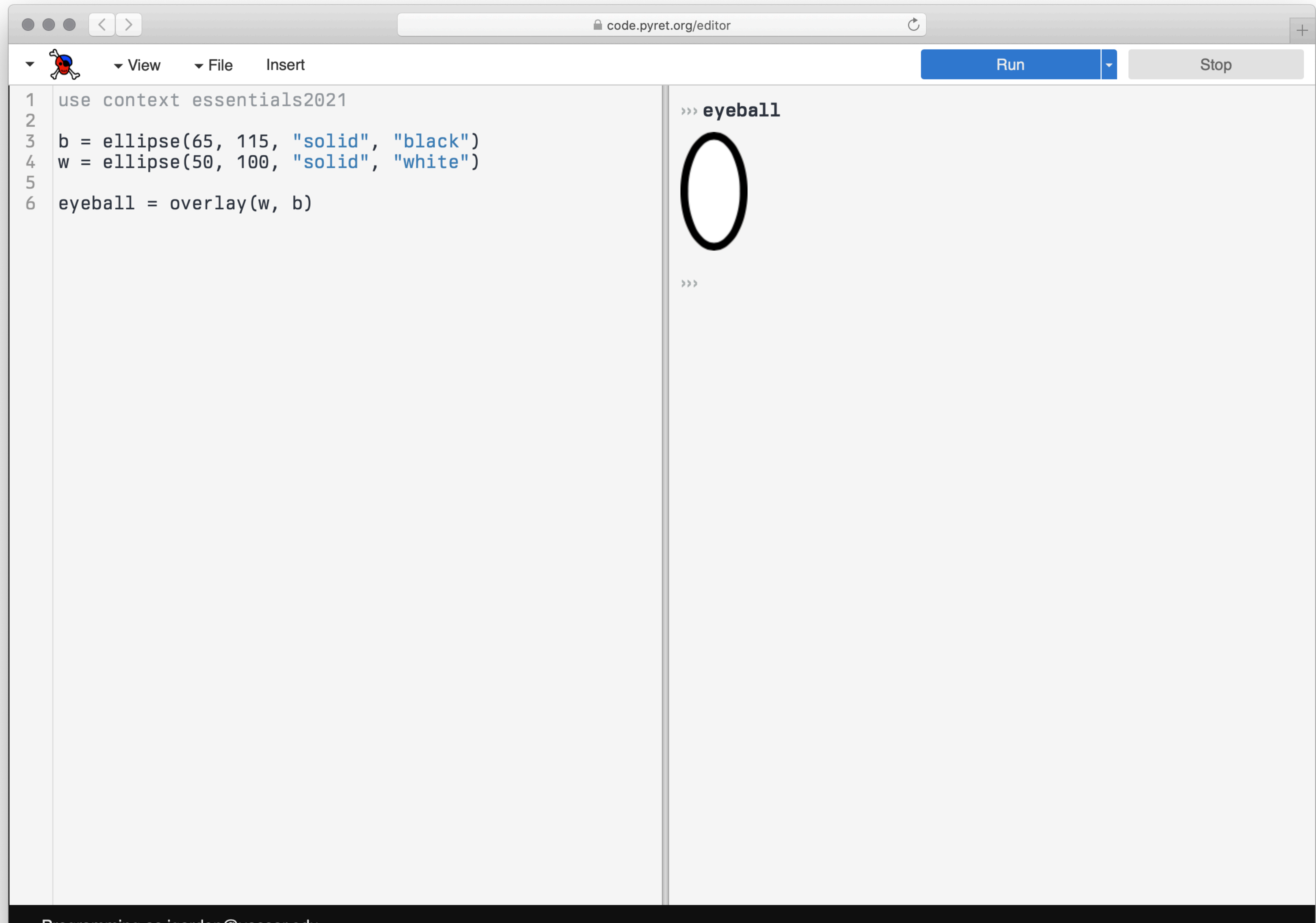
xeyes



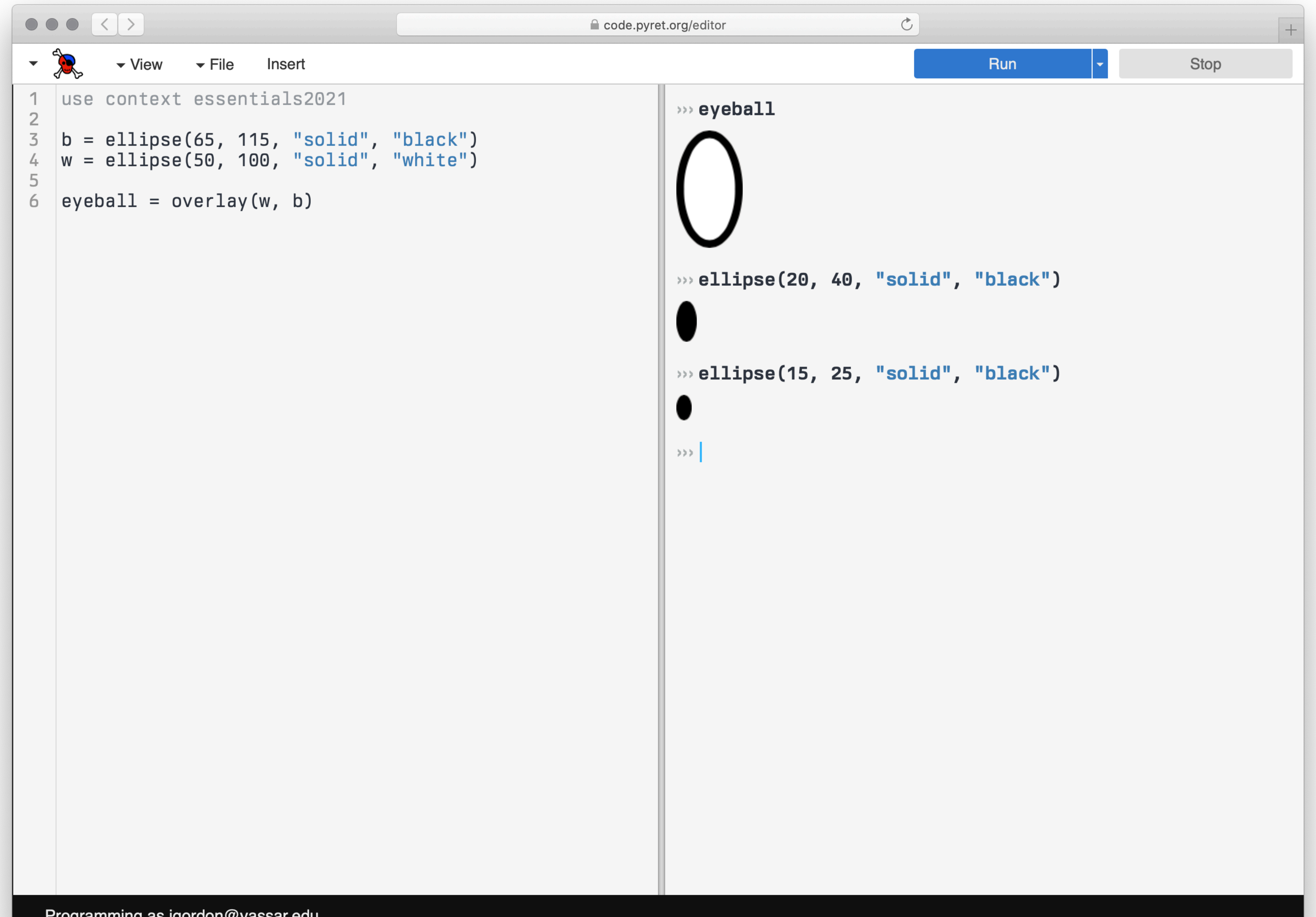
xeyes

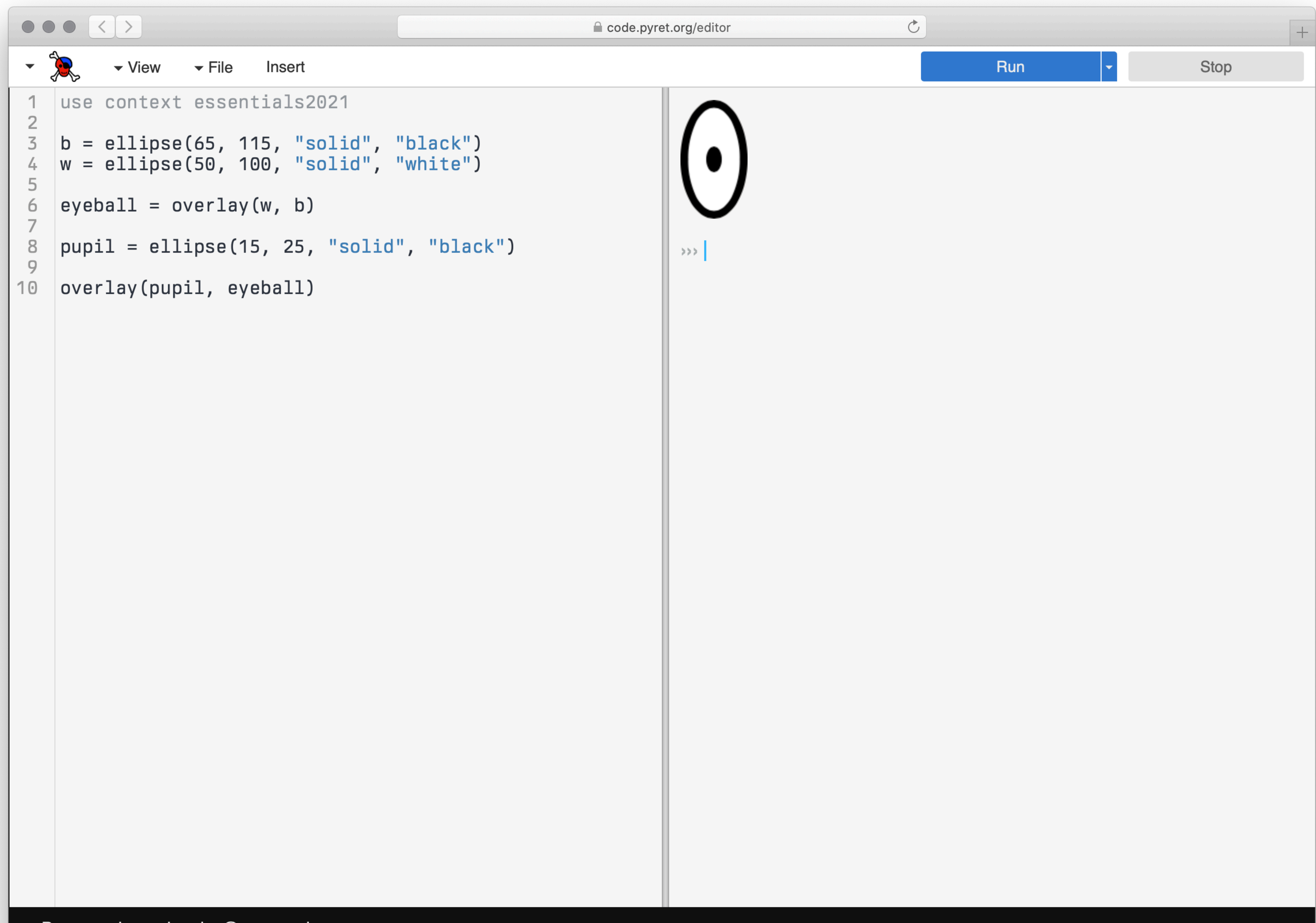


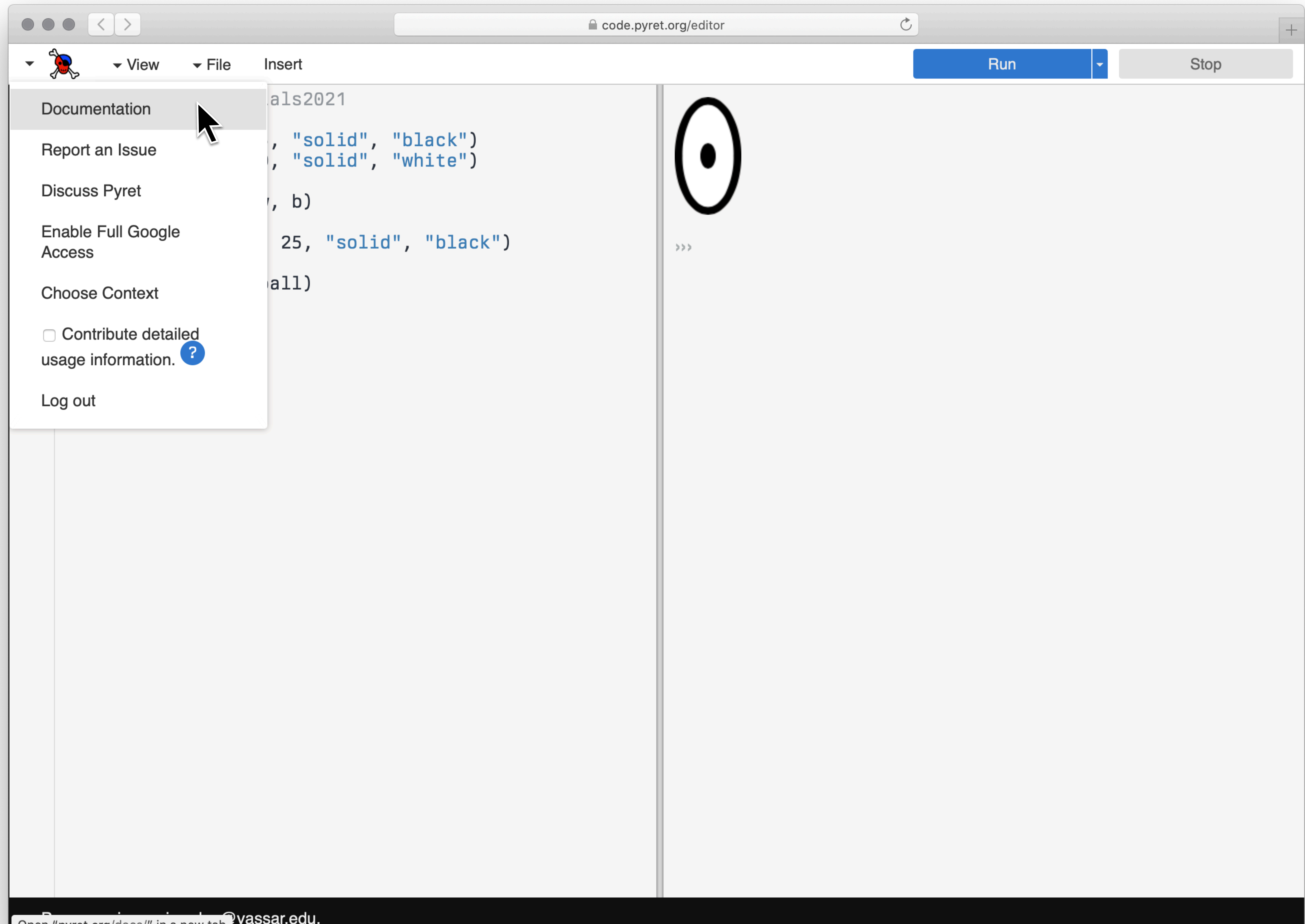

code.pyret.org/editor
View
File
Insert
Run
Stop
1 use context essentials2021
2
››› include image
››› ellipse(30, 60, "outline", "black")
››› ellipse(50, 80, "outline", "black")
››› ellipse(70, 100, "outline", "black")
››› ellipse(70, 110, "outline", "black")
›››
Programming as jgordon@vassar.edu.



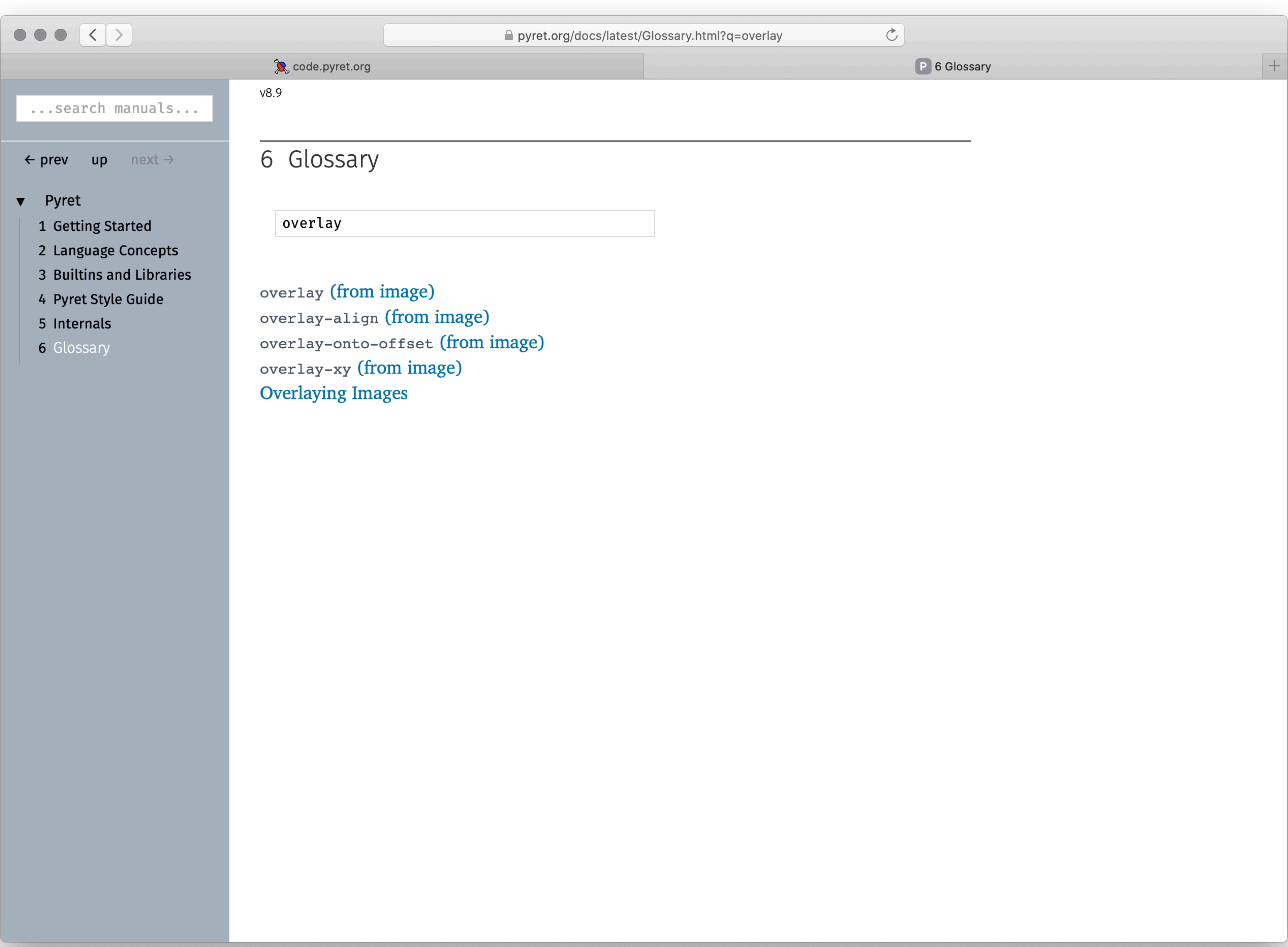
code.pyret.org/editor
View
File
Insert
Run
Stop
1 use context essentials2021
2
3 b = ellipse(65, 115, "solid", "black")
4 w = ellipse(50, 100, "solid", "white")
5
6 eyeball = overlay(w, b)
››› eyeball
›››
Programming as jgordon@vassar.edu.


code.pyret.org/editor
View
File
Insert
Run
Stop
1 use context essentials2021
2
3 b = ellipse(65, 115, "solid", "black")
4 w = ellipse(50, 100, "solid", "white")
5
6 eyeball = overlay(w, b)
››› eyeball
››› ellipse(20, 40, "solid", "black")
››› ellipse(15, 25, "solid", "black")
›››
Programming as jgordon@vassar.edu.


code.pyret.org/editor
View
File
Insert
Run
Stop
1 use context essentials2021
2
3 b = ellipse(65, 115, "solid", "black")
4 w = ellipse(50, 100, "solid", "white")
5
6 eyeball = overlay(w, b)
7
8 pupil = ellipse(15, 25, "solid", "black")
9
10 overlay(pupil, eyeball)
›››
Programming as jgordon@vassar.edu.



code.pyret.org/editor
View
File
Insert
Run
Stop
Documentation
Report an Issue
Discuss Pyret
Enable Full Google Access
Choose Context
Contribute detailed usage information.
Log out
als2021
, "solid", "black")
, "solid", "white")
, b)
25, "solid", "black")
all)
@vassar.edu.
Open "pyret.org/docs/" in a new tab


pyret.org/docs/latest/Glossary.html?q=overlay
code.pyret.org
6 Glossary
...search manuals...
v8.9
← prev
up
next →
6 Glossary
overlay
▼ Pyret
1 Getting Started
2 Language Concepts
3 Builtins and Libraries
4 Pyret Style Guide
5 Internals
6 Glossary
overlay (from image)
overlay-align (from image)
overlay-onto-offset (from image)
overlay-xy (from image)
Overlaying Images

...search manuals...
← prev up next →
▶ Pyret
▼ 3 Builtins and Libraries
3.1 Global Utilities
3.2 Numbers
3.3 Strings
3.4 Booleans
3.5 RawArray
3.6 Tables
3.7 lists
3.8 sets
3.9 arrays
3.10 string-dict
3.11 option
3.12 pick
3.13 either
3.14 srcloc
3.15 pprint
3.16 s-exp
3.17 s-exp-structs
3.18 color
3.19 image-structs
3.20 The image libraries
3.21 world
3.22 gdrive-sheets
3.23 data-source
3.24 reactors
3.25 chart
3.26 plot
3.27 statistics

```
overlay :: (
   img1 :: Image,
   img2 :: Image
)
-> Image
```

Constructs a new image where `img1` overlays `img2`. The two images are aligned at their pinholes, so `overlay(img1, img2)` behaves like `overlay-align("pinhole", "pinhole", img1, img2)`.

Examples:

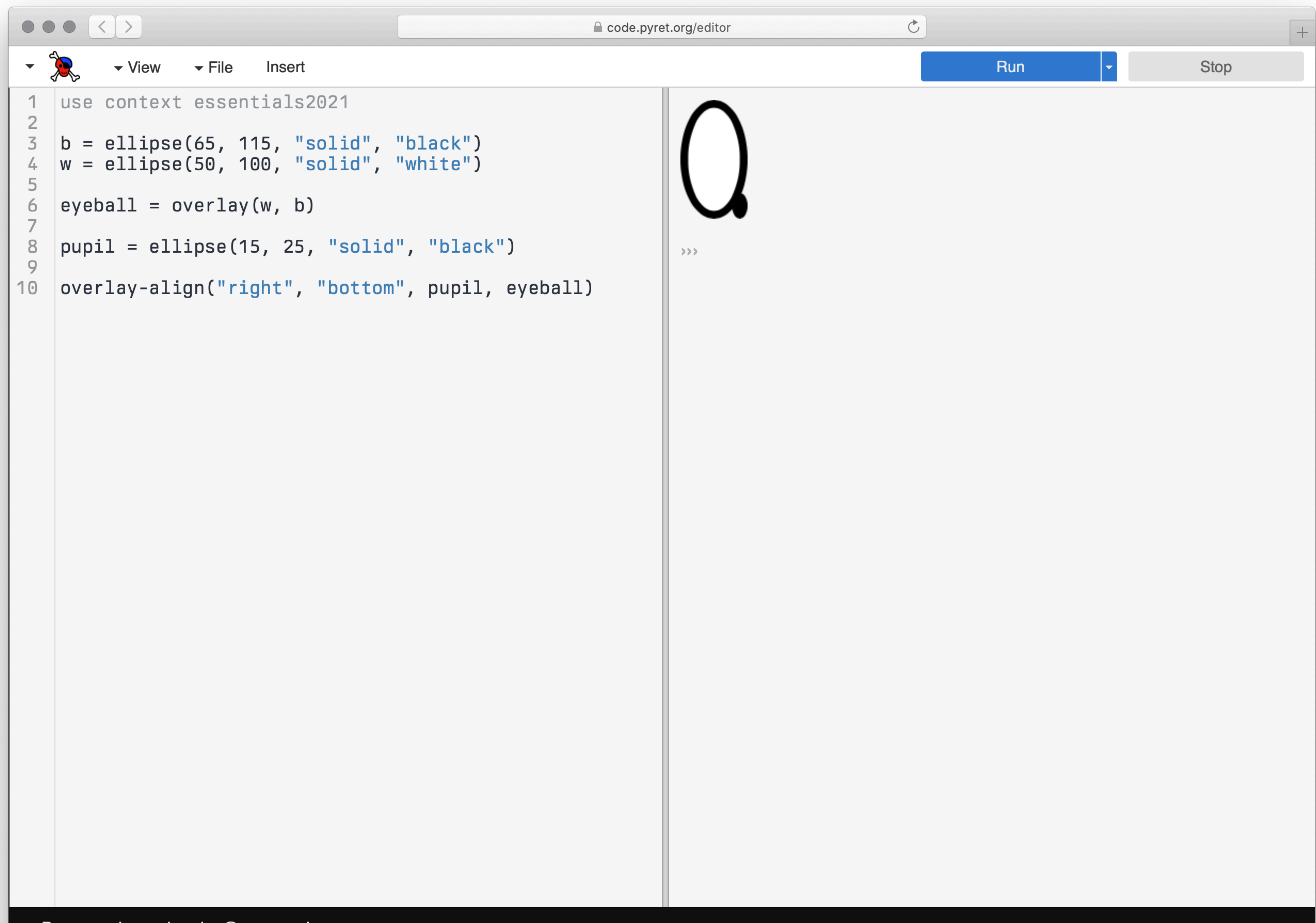
```
››› overlay(rectangle(30, 60, "solid", "orange"),
      ellipse(60, 30, "solid", "purple"))
```

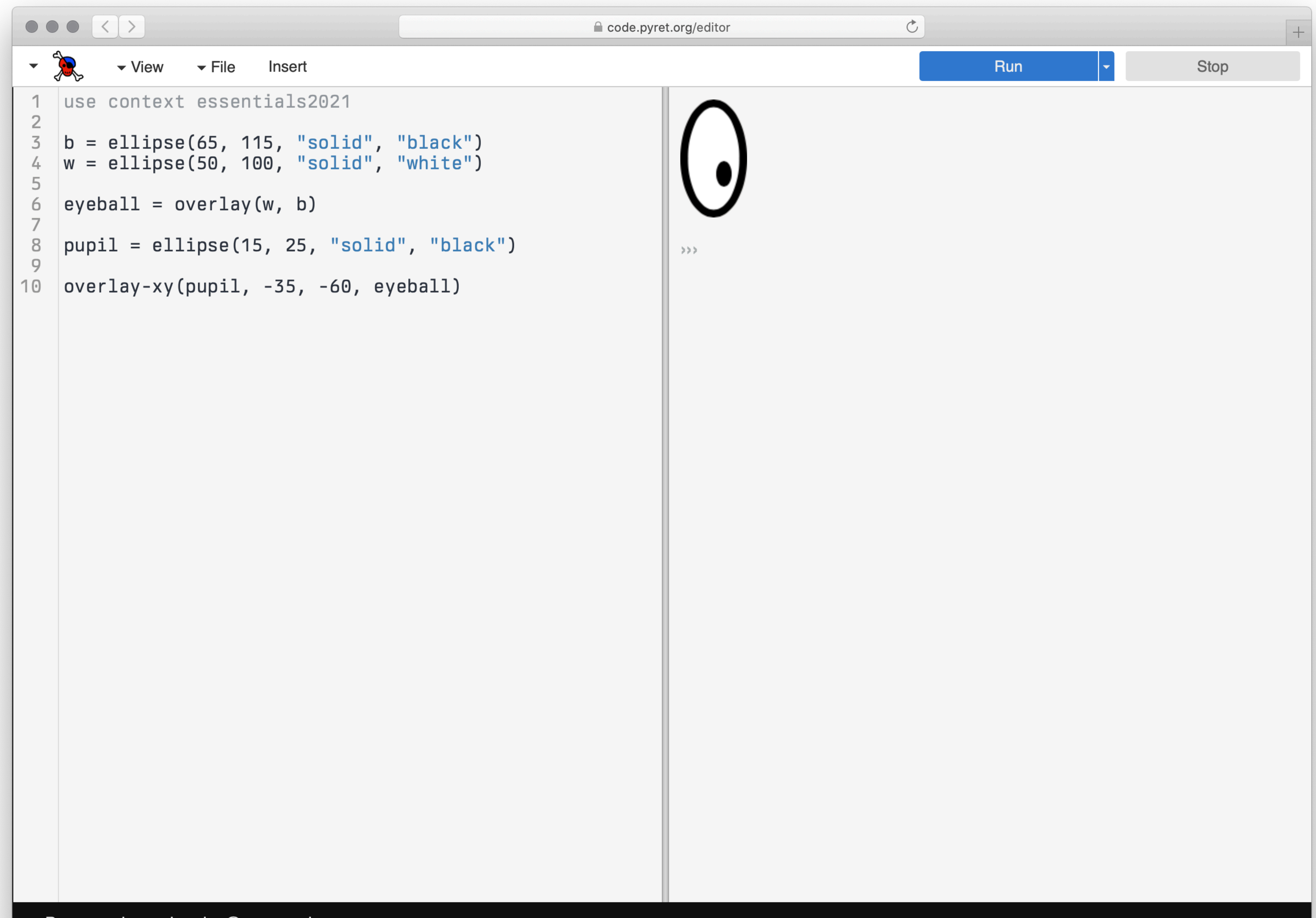
```
overlay-align :: (
   place-x :: XPlace,
   place-y :: YPlace,
   img1 :: Image,
   img2 :: Image
)
-> Image
```

Overlays `img1` on `img2` like `overlay`, but uses `place-x` and `place-y` to determine the alignment point in each image. A call to `overlay-align(place-x, place-y, img1, img2)` behaves the same as `overlay-onto-offset(img1, place-x, place-y, 0, 0, img2, place-x, place-y)`

Examples:

```
››› overlay-align("left", "bottom",
      square(30, "solid", "bisque"), square(50, "solid", "dark-green"))
```


code.pyret.org/editor
View
File
Insert
Run
Stop
use context essentials2021
b = ellipse(65, 115, "solid", "black")
w = ellipse(50, 100, "solid", "white")
eyeball = overlay(w, b)
pupil = ellipse(15, 25, "solid", "black")
overlay-align("right", "bottom", pupil, eyeball)
›››
Programming as jgordon@vassar.edu.


code.pyret.org/editor
View
File
Insert
Run
Stop
1 use context essentials2021
2
3 b = ellipse(65, 115, "solid", "black")
4 w = ellipse(50, 100, "solid", "white")
5
6 eyeball = overlay(w, b)
7
8 pupil = ellipse(15, 25, "solid", "black")
9
10 overlay-xy(pupil, -35, -60, eyeball)
›››
Programming as jgordon@vassar.edu.

code.pyret.org/editor
View
File
Insert
Run
Stop
use context essentials2021
b = ellipse(65, 115, "solid", "black")
w = ellipse(50, 100, "solid", "white")
eyeball = overlay(w, b)
pupil = ellipse(15, 25, "solid", "black")
left-eye = overlay-xy(pupil, -35, -60, eyeball)
right-eye = flip-horizontal(left-eye)
beside(left-eye, right-eye)
Programming as jgordon@vassar.edu.

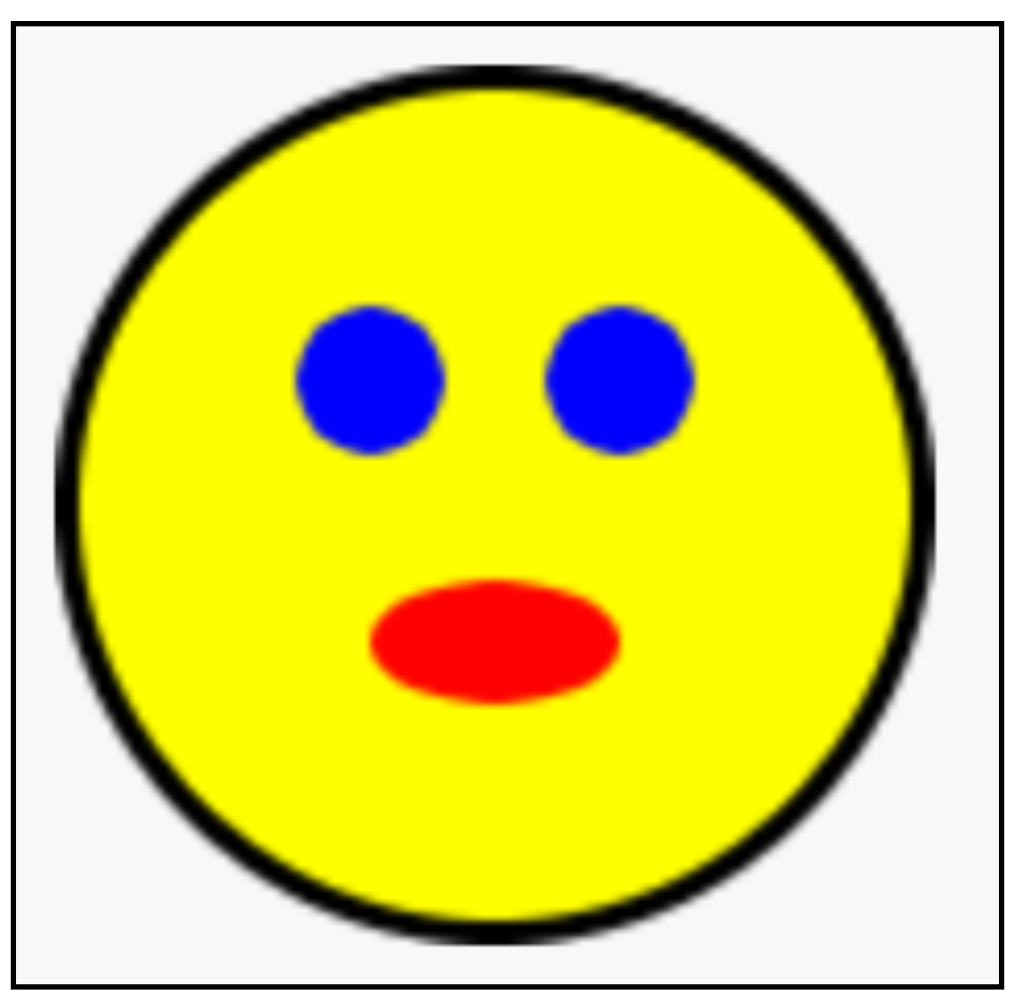
As you build up more complex images from simpler ones, you're following a core idea called *composition*.

Programs are always built of smaller programs that do parts of the larger task you want to perform.

We'll use composition throughout this course.

Organizing a program with names

Let’s consider three programs that all draw this (beautiful, nuanced) emoji:



Version 1

```
# Create the head: a yellow circle with black border
base = circle(50, "solid", "yellow")
base-border = circle(53, "solid", "black")
head = overlay(base, base-border)

# Create pair of eyes, using a square as a spacer
eye = circle(9, "solid", "blue")
eye-spacer = square(12, "solid", "yellow")
one-eye-with-space = beside(eye, eye-spacer)
eyes = beside(one-eye-with-space, eye)

# Add a mouth to the eyes to make a face
mouth = ellipse(30, 15, "solid", "red")
mouth-spacer = rectangle(30, 15, "solid", "yellow")
eyes-with-mouth-space = above(eyes, mouth-spacer)
face = above(eyes-with-mouth-space, mouth)

# Put the face on the head
emoji = overlay-align("center", "center", face, head)
emoji
```

Version 2

```
# Create the head: a yellow circle with black border
base = circle(50, "solid", "yellow")
head = overlay(base, circle(53, "solid", "black"))

# Create a pair of eyes, using a square as a spacer
eye = circle(9, "solid", "blue")
eyes =
  beside(
    eye,
    beside(
      square(12, "solid", "yellow"),  # eye spacer
      eye))

# Add a mouth to the eyes to make a face
mouth = ellipse(30, 15, "solid", "red")
face =
  above(
    eyes,
    above(
      rectangle(30, 15, "solid", "yellow"),  # mouth spacer
      mouth))

# Put the face on the head
emoji = overlay-align("center", "center", face, head)
emoji
```

Version 3

```
overlay-align("center", "center",
  above(
    beside(
      circle(9, "solid", "blue"),  # eye
      beside(
        square(12, "solid", "yellow"),  # eye spacer
        circle(9, "solid", "blue"))),  # eye
    above(
      rectangle(30, 15, "solid", "yellow"),  # mouth spacer
      ellipse(30, 15, "solid", "red"))),  # mouth
  overlay(circle(50, "solid", "yellow"),  # base
    circle(53, "solid", "black")))  # head border
```

All three programs generate the same image.

Which one seems easiest to read and understand?

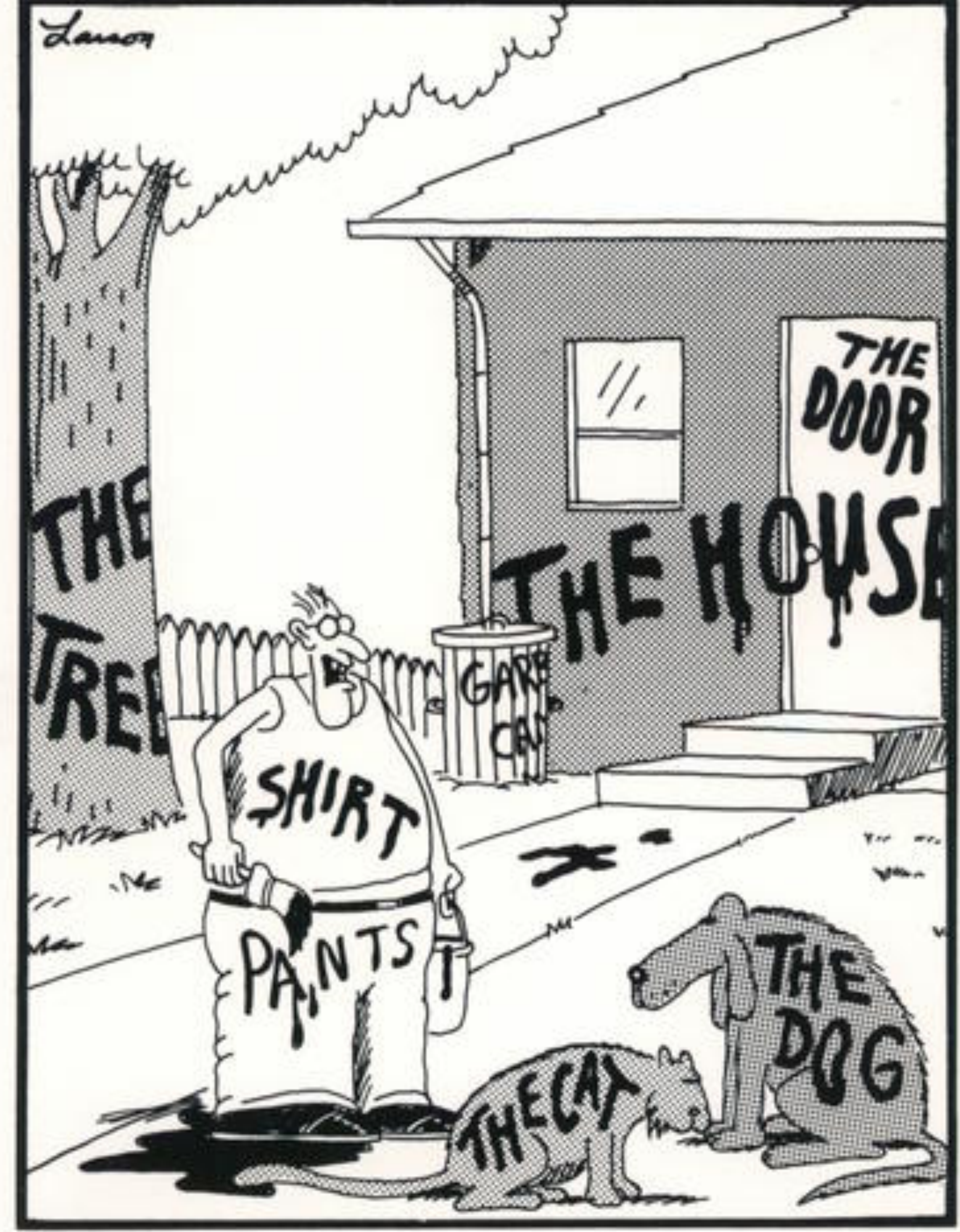
Version 1

```
# Create the head: a yellow circle with black border
base = circle(50, "solid", "yellow")
base-border = circle(53, "solid", "black")
head = overlay(base, base-border)

# Create pair of eyes, using a square as a spacer
eye = circle(9, "solid", "blue")
eye-spacer = square(12, "solid", "yellow")
one-eye-with-space = beside(eye, eye-spacer)
eyes = beside(one-eye-with-space, eye)

# Add a mouth to the eyes to make a face
mouth = ellipse(30, 15, "solid", "red")
mouth-spacer = rectangle(30, 15, "solid", "yellow")
eyes-with-mouth-space = above(eyes, mouth-spacer)
face = above(eyes-with-mouth-space, mouth)

# Put the face on the head
emoji = overlay-align("center", "center", face, head
emoji
```



"Now! . . . *That* should clear up a few things around here!"

Version 3

```
overlay-align("center", "center",
  above(
    beside(
      circle(9, "solid", "blue"),  # eye
      beside(
        square(12, "solid", "yellow"),  # eye spacer
        circle(9, "solid", "blue"))),  # eye
    above(
      rectangle(30, 15, "solid", "yellow"),  # mouth spacer
      ellipse(30, 15, "solid", "red"))),  # mouth
  overlay(circle(50, "solid", "yellow"),  # base
    circle(53, "solid", "black")))  # head border
```

Version 3

```
overlay-align("center", "center",
  above(
    beside(
      circle(9, "solid", "blue"),
      beside(
        square(12, "solid", "yellow"),
        circle(9, "solid", "blue"))),
    above(
      rectangle(30, 15, "solid", "yellow"),
      ellipse(30, 15, "solid", "red"))),
  overlay(circle(50, "solid", "yellow"),
    circle(53, "solid", "black")))
```

Version 2

```
# Create the head: a yellow circle with black border
base = circle(50, "solid", "yellow")
head = overlay(base, circle(53, "solid", "black"))

# Create a pair of eyes, using a square as a spacer
eye = circle(9, "solid", "blue")
eyes =
  beside(
    eye,
    beside(
      square(12, "solid", "yellow"),  # eye spacer
      eye))

# Add a mouth to the eyes to make a face
mouth = ellipse(30, 15, "solid", "red")
face =
  above(
    eyes,
    above(
      rectangle(30, 15, "solid", "yellow"),  # mouth spacer
      mouth))

# Put the face on the head
emoji = overlay-align("center", "center", face, head)
emoji
```

Gemma Correll

Beginning programmers tend to write code more like the first or third examples, naming everything or nothing.

As we get more into working with structured data, writing code like the second example will be useful, as the structure of a well-written program tends to reflect the structure of the data you are working with.

“Programs must be written for people to read, and only incidentally for machines to execute.”

Hal Abelson & Gerald Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, 1979

Acknowledgments

This class incorporates material from:

Kathi Fisler, Brown University

Jason Waterman, Vassar College