CMPU 101 §02 · Computer Science I

Expressions, Values, and Names

31 August 2022

Where are we?

A *program* (or *script*) instructs a computer to do something.

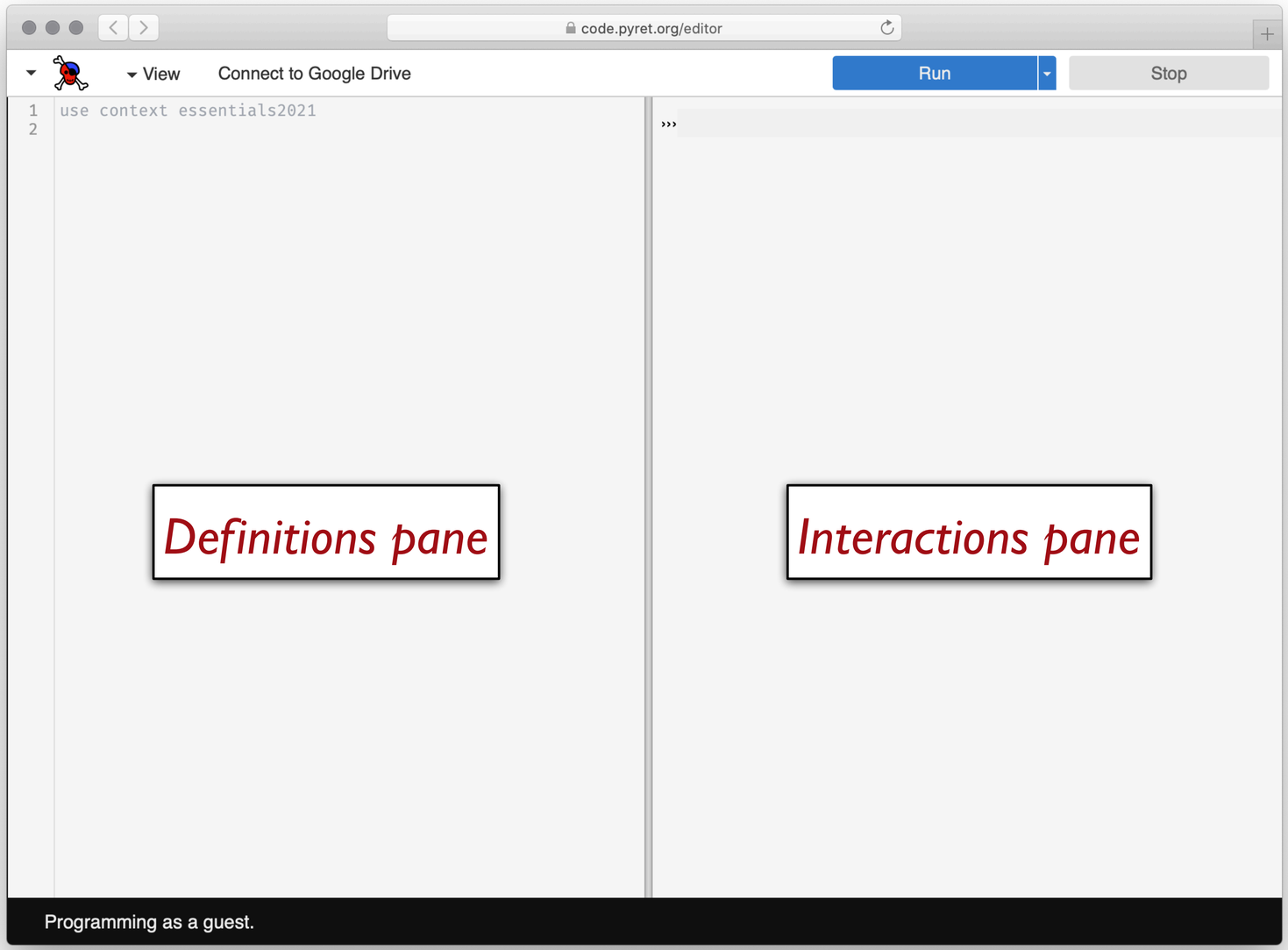
These instructions must be very specific for the computer to carry them out.

But programs also need to be understood by people, so they must be readable!

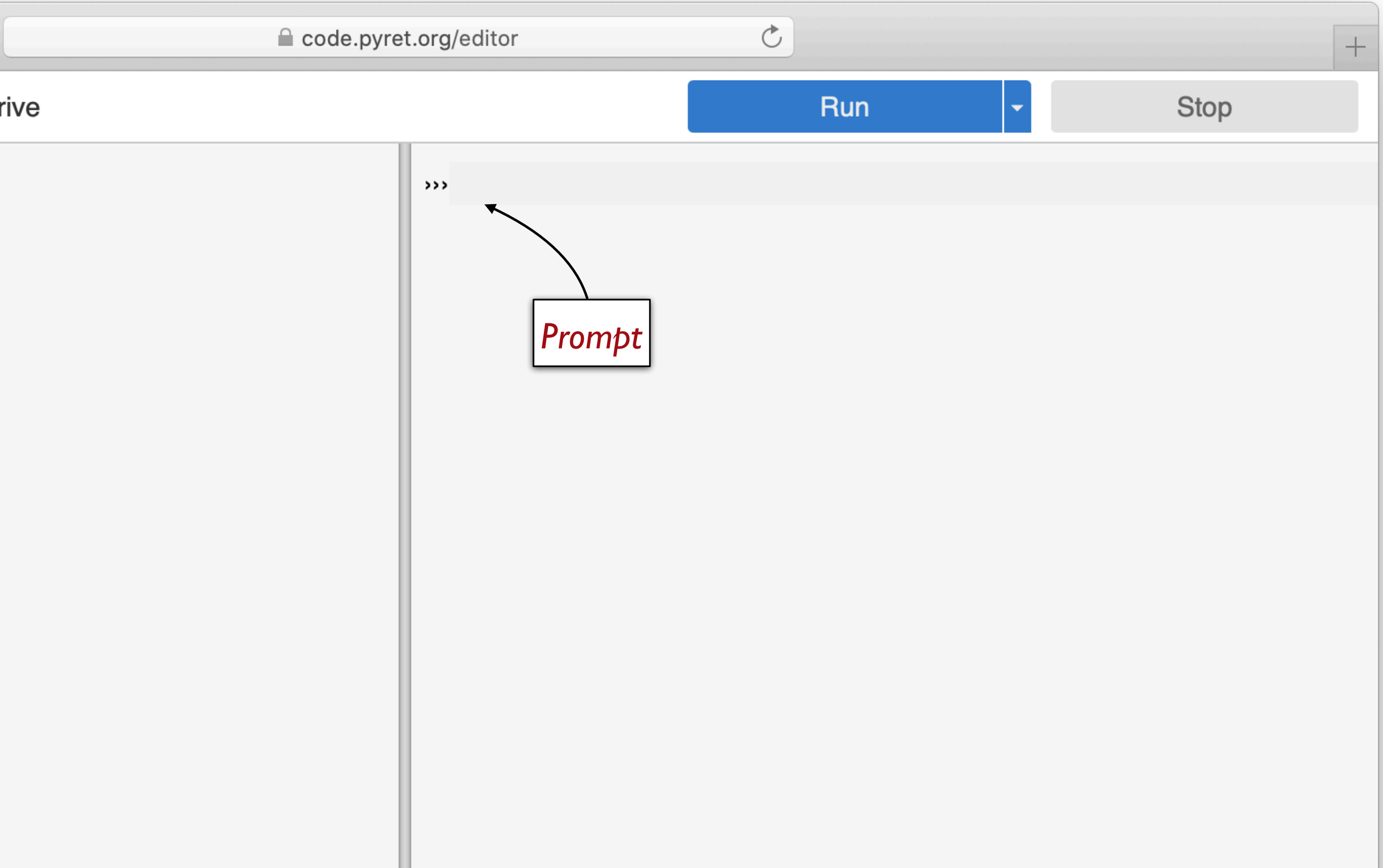
To write a program, we need to use a *programming language* and *programming environment*.

We write our computation in the language.

We run the program in the environment.



code.pyret.org


code.pyret.org/editor
rive
Run
Stop
>>>
Prompt

Use the *interactions pane* for:

- Trying out expressions
- Checking syntax

Use the *definitions pane* for:

- Building complex expressions
- Naming expressions
- Using previously defined expressions
- Saving your code as files!

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?

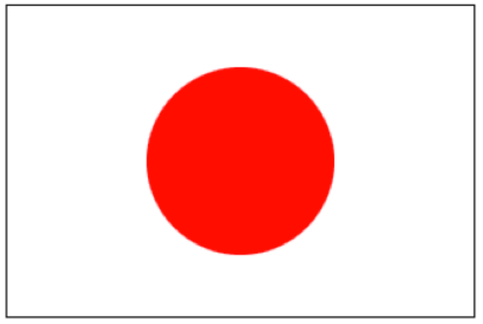
Starting to program

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 consists of one or more computations you want to run.

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

Reading this expression errored:

interactions://1:0:0-0:9

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

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)
23
>>>
```

```
>>> 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.

```
>>> 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.

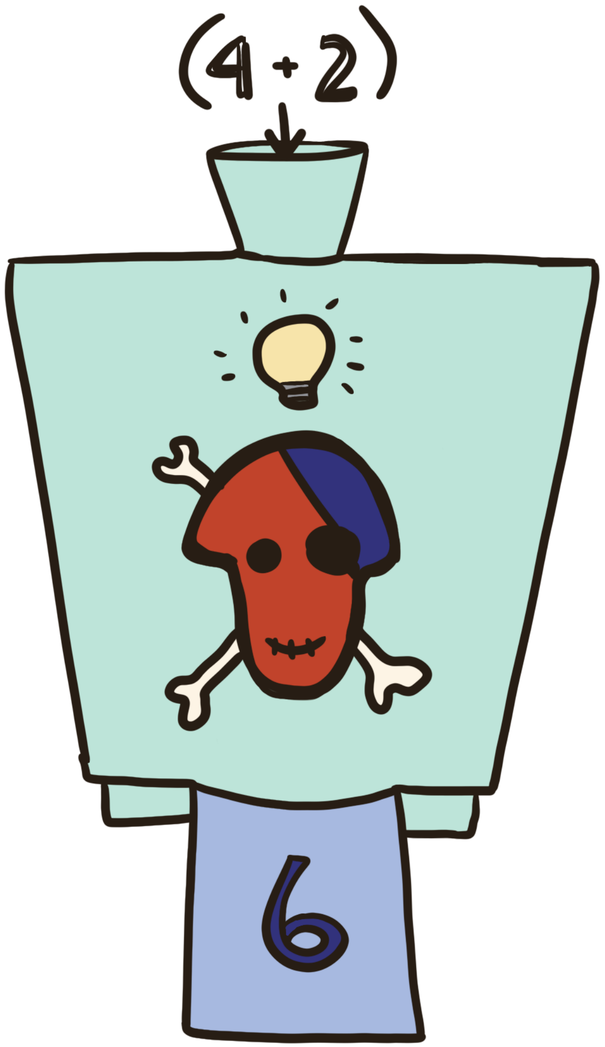
Evaluation

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

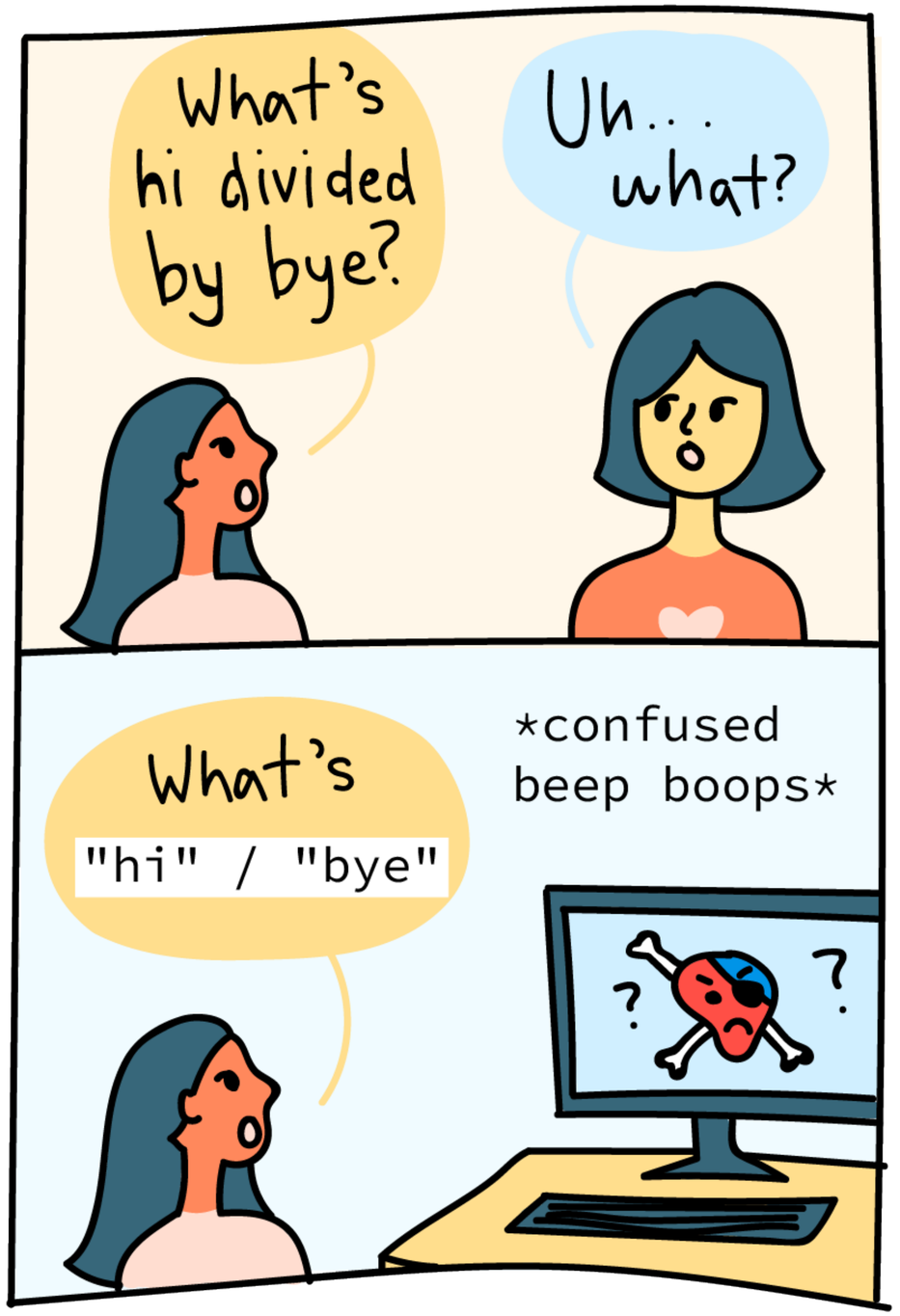
- What is the operator `/` dividing?
- Shouldn't `/` expect two numbers?

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!



What’s in a name?

An expression of the form

⟨***name***⟩ = ⟨***expression***⟩

tells Pyret to associate the value of ⟨***expression***⟩ with ⟨***name***⟩.

Every time you type ⟨***name***⟩, Pyret will substitute the value for you, e.g.,

```
x = 5
x + 4
```

will evaluate to 9.

Note there's no output from entering a definition.

It only has a side effect of telling Pyret to associate the name with the value.

```
››› star(40, "solid", "gold")
```



```
››› my-star = star(40, "solid", "gold")
››› my-star
```

To evaluate a definition,

1. Evaluate the expression and record the resulting value as the value of the name

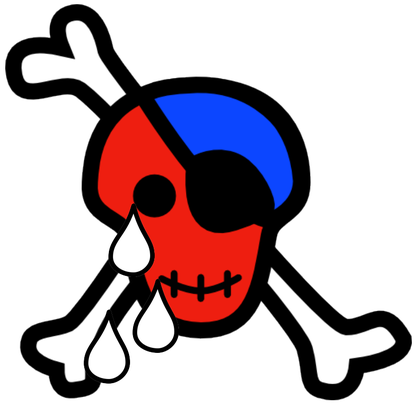
To evaluate a defined name,

1. Lookup the value associated with the name

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
```



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
>>> sept = 7
>>> sept
7
```

If we define constants

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

Now if we write

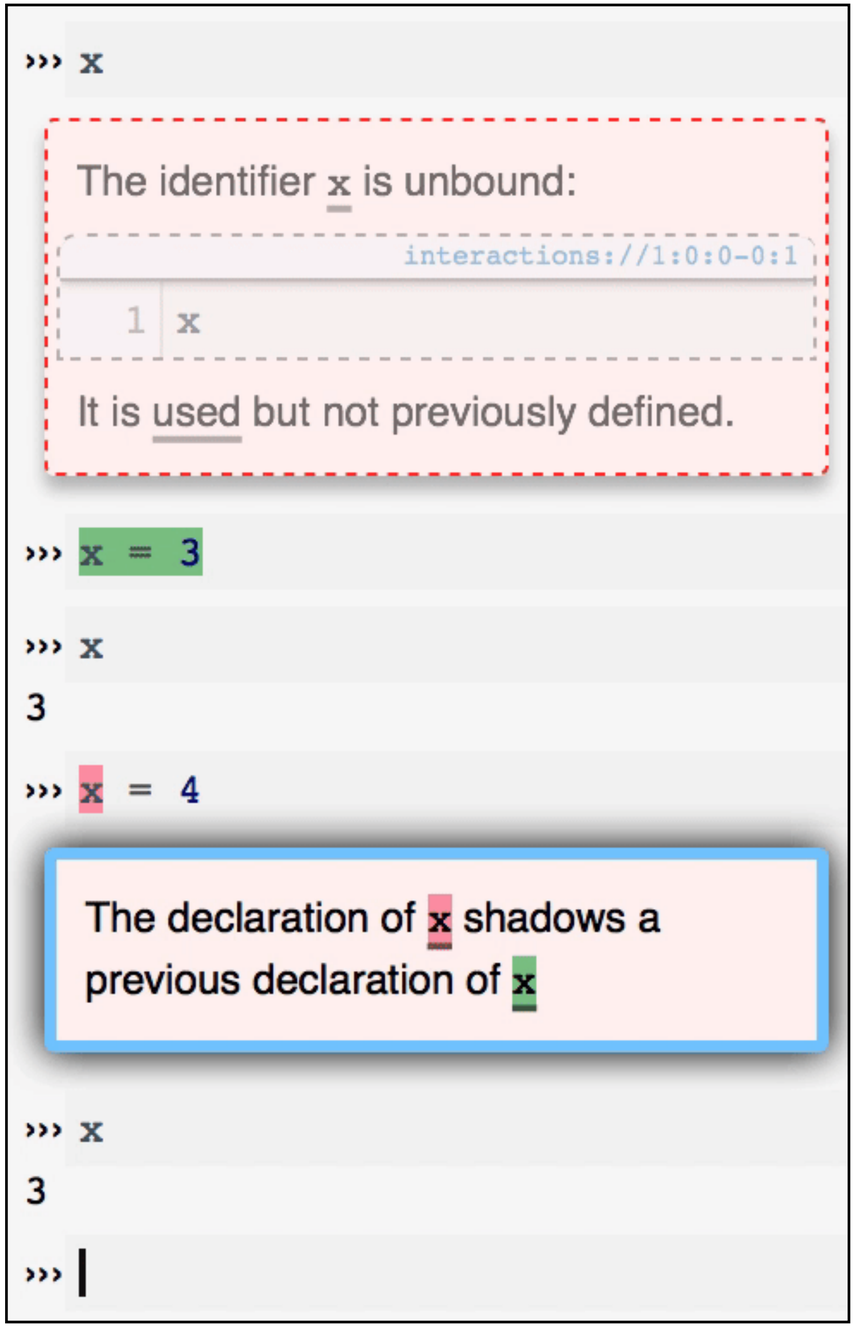
```
width * height
```

it gets evaluated:

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

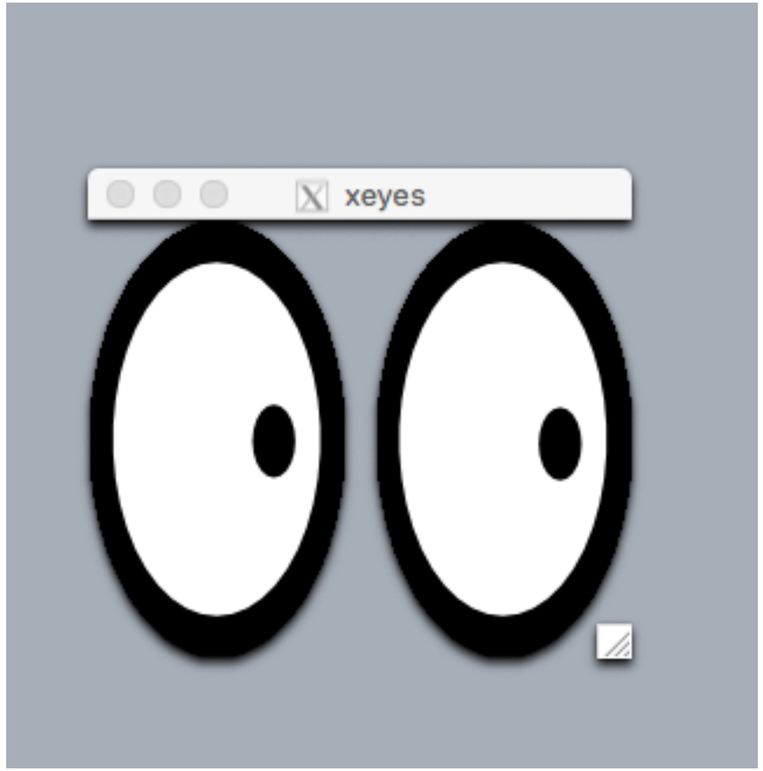
Names must be given a value before being used.

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

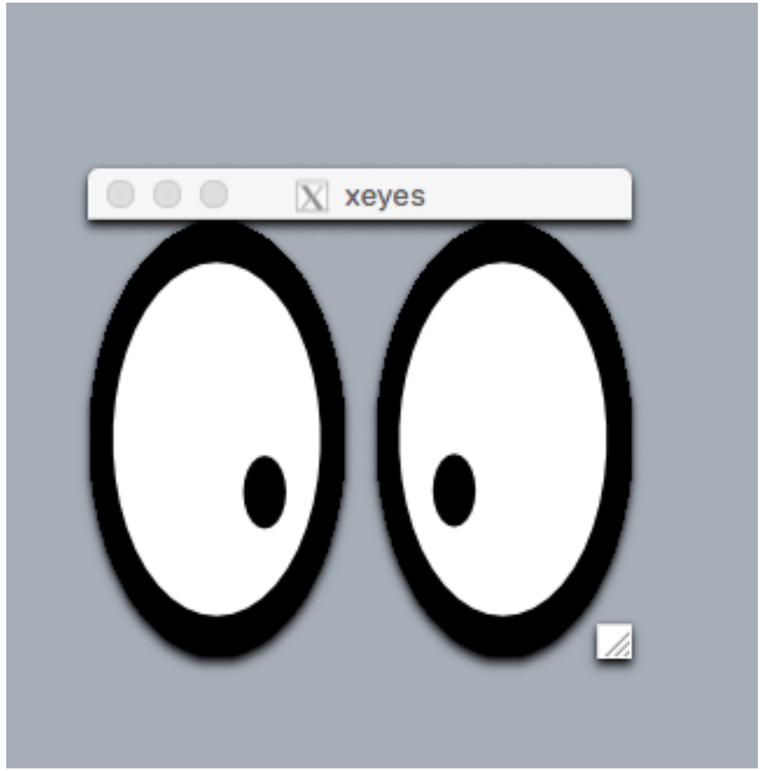


Exercise

xeyes



xeyes



code.pyret.org/editor

View Connect to Google Drive 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 a guest.

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  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 eyeball = overlay(w, b)
```

```
››› eyeball
```

```
››› ellipse(20, 40, "solid", "black")
```

```
››› ellipse(15, 25, "solid", "black")
```

```
›››
```

Programming as jgordon@vassar.edu.

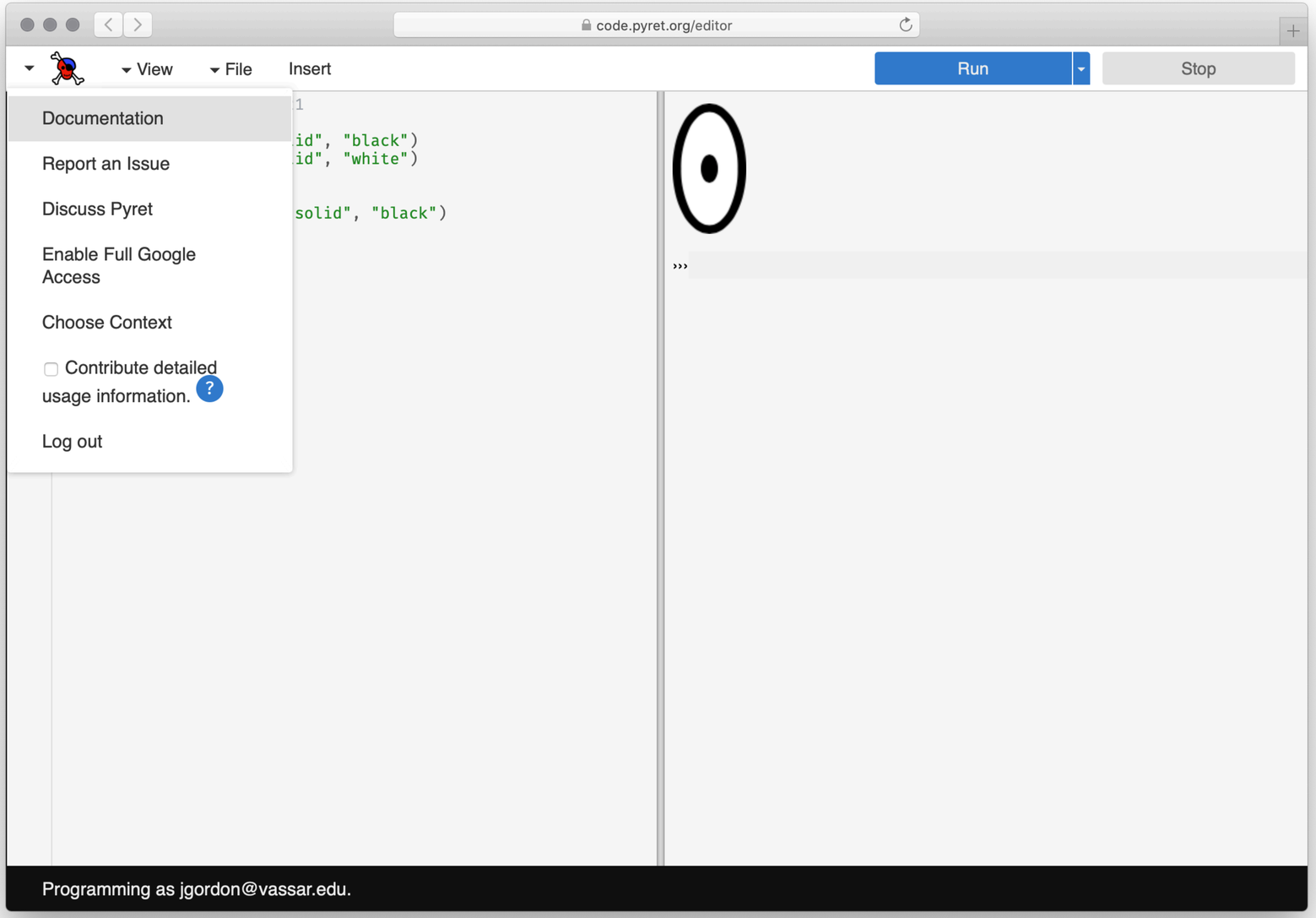
code.pyret.org/editor

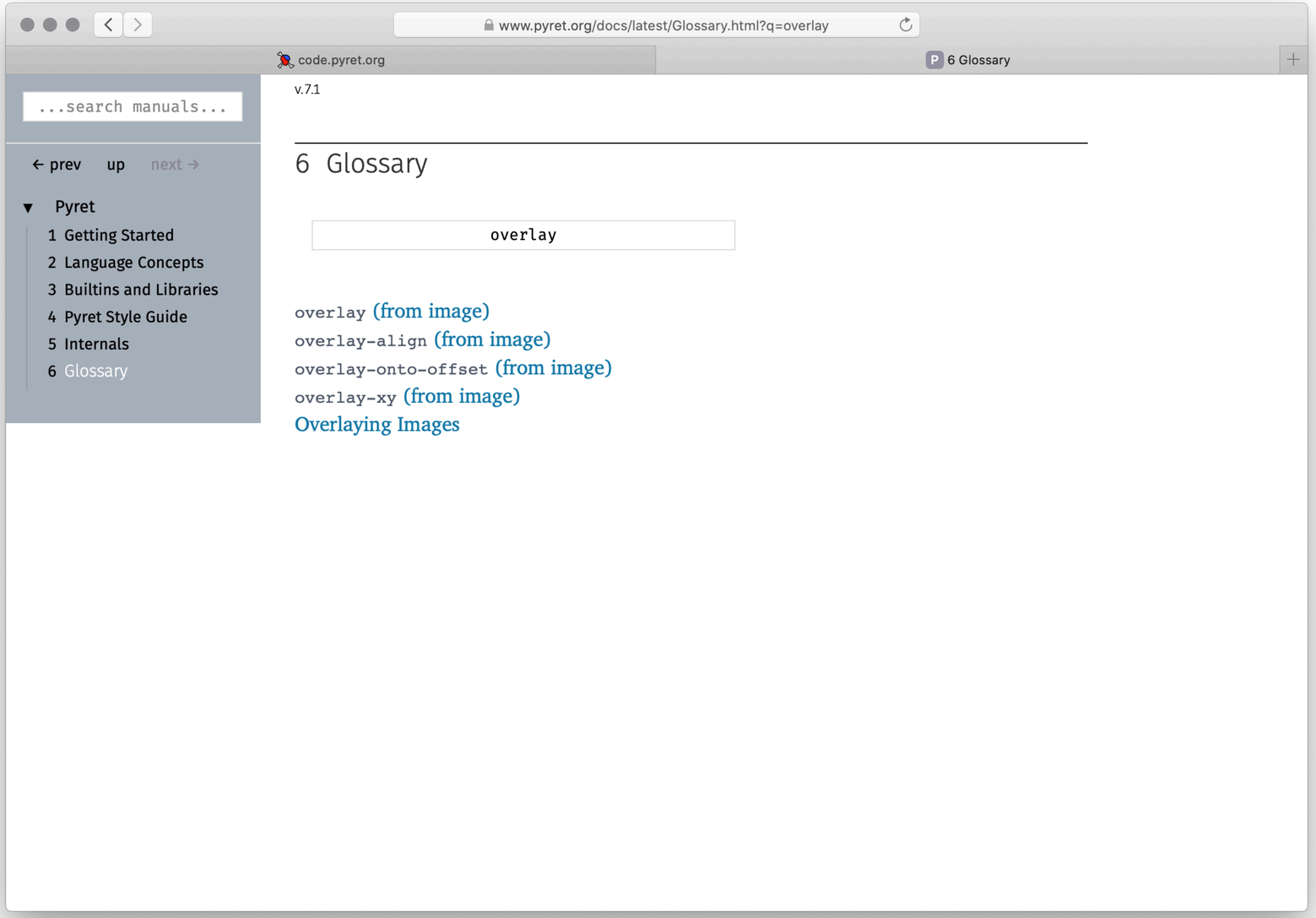
View File Insert Run Stop

```
1  use context essentials2021
2
3  b = ellipse(65, 115, "solid", "black")
4  w = ellipse(50, 100, "solid", "white")
5  eyeball = overlay(w, b)
6
7  pupil = ellipse(15, 25, "solid", "black")
8
9  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
1
id", "black")
id", "white")
solid", "black")
›››
Programming as jgordon@vassar.edu.


www.pyret.org/docs/latest/Glossary.html?q=overlay
code.pyret.org
6 Glossary
...search manuals...
← prev up next →
▼ Pyret
1 Getting Started
2 Language Concepts
3 Builtins and Libraries
4 Pyret Style Guide
5 Internals
6 Glossary
v.7.1
6 Glossary
overlay
overlay (from image)
overlay-align (from image)
overlay-onto-offset (from image)
overlay-xy (from image)
Overlaying Images

```
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)`

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  eyeball = overlay(w, b)
6
7  pupil = ellipse(15, 25, "solid", "black")
8
9  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  eyeball = overlay(w, b)
6
7  pupil = ellipse(15, 25, "solid", "black")
8
9  overlay-xy(pupil, -35, -60, eyeball)
```

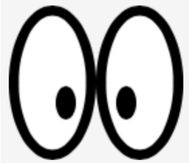
›››

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  eyeball = overlay(w, b)
6
7  pupil = ellipse(15, 25, "solid", "black")
8
9  left-eye = overlay-xy(pupil, -35, -60, eyeball)
10 right-eye = flip-horizontal(left-eye)
11
12 beside(left-eye, right-eye)
```



›››

Programming as jgordon@vassar.edu.

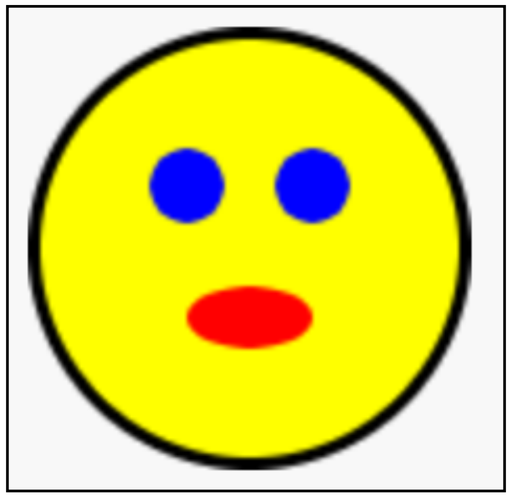
As you build up more complex images from simpler ones, you're following a core idea called *composition*.

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
```

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 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
```

Beginning programmers tend to write code more like the first example.

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

Acknowledgments

This class incorporates material from:

Kathi Fisler, Brown University

Jason Waterman, Vassar College

Jonathan Gordon, Vassar College