Expressions, Values, and Names

24 January 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.
use context essentials2021
Definitions pane

Interactions pane

code.pyret.org
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…
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
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 \textit{numbers}.

We need a way to describe \textit{colors} to our program.

We need a way to create images based on simple \textit{shapes} of different colors.
An individual number like 5 is a value – it can’t be computed any further.
\((3 + 4) \times (5 + 1)\) is an \textit{expression} – a computation that produces an answer.

A program consists of one or more computations you want to run.
Reading this expression errored:

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

```python
>>> 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.
>>> include image

>>> circle(50, "solid", "red")
We can manipulate images much like we can manipulate numbers.

- Numbers can be added, subtracted, etc.
- Images can be overlaid, rotated, flipped, etc.
Evaluation
How does something like $\frac{4 + 2}{3}$ 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

\[ \langle name \rangle = \langle expression \rangle \]

tells Pyret to associate the value of \( \langle expression \rangle \) with \( \langle name \rangle \).

Every time you type \( \langle name \rangle \), Pyret will substitute the value for you, e.g.,

\[
\begin{align*}
x &= 5 \\
x + 4
\end{align*}
\]

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.
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.,

```python
this-is-a-good-name
this_makes_bonny_cry
thisIsACrimeAgainstPyret
```
Names are arbitrary

The following is silly, but legal:

```python
>>> five = 6
>>> five
6
>>> six = 5
>>> six
5
```
Several constants may have the same value:

```python
>>> seven = 7
>>> seven
7
>>> sept = 7
>>> sept
7
```
If we define constants

\[
\begin{align*}
\text{width} & = 400 \\
\text{height} & = 600
\end{align*}
\]

Now if we write

\[
\text{width} \times \text{height}
\]

it gets evaluated:

\[
\begin{align*}
\rightarrow & \quad 400 \quad \times \quad \text{height} \\
\rightarrow & \quad 400 \quad \times \quad 600 \\
\rightarrow & \quad 240000
\end{align*}
\]
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.
Exercise
xeyes
xeyes
```cpp
use context essentials2021

#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.
use context essentials2021
b = ellipse(65, 115, "solid", "black")
w = ellipse(50, 100, "solid", "white")
eyeball = overlay(w, b)
use context essentials2021

b = ellipse(65, 115, "solid", "black")
w = ellipse(50, 100, "solid", "white")
eyeball = overlay(w, b)

eyeball

circle(20, 40, "solid", "black")

circle(15, 25, "solid", "black")

...
use context essentials2021
b = ellipse(65, 115, "solid", "black")
w = ellipse(58, 100, "solid", "white")
eyeball = overlay(w, b)
pupil = ellipse(15, 25, "solid", "black")
overlay(pupil, eyeball)
6 Glossary

overland

overland (from image)
overland-align (from image)
overland-onto-offset (from image)
overland-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)
use context essentials2021
b = ellipse(65, 115, "solid", "black")
w = ellipse(58, 100, "solid", "white")
eyeball = overlay(w, b)
pupil = ellipse(15, 25, "solid", "black")
overlay-align("right", "bottom", pupil, eyeball)
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-xy(pupil, -35, -60, eyeball)
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)
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:
# 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 1
# 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
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?
# 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
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
# Create the head: a yellow circle with black border

```plaintext
def base = circle(50, "solid", "yellow")
def head = overlay(base, circle(53, "solid", "black"))
```

# Create a pair of eyes, using a square as a spacer

```plaintext
def eye = circle(9, "solid", "blue")
def eyes =
    beside(
        eye,
        beside(
            square(12, "solid", "yellow"),  # eye spacer
            eye))
```

# Add a mouth to the eyes to make a face

```plaintext
def mouth = ellipse(30, 15, "solid", "red")
def face =
    above(
        eyes,
        above(
            rectangle(30, 15, "solid", "yellow"),  # mouth spacer
            mouth))
```

# Put the face on the head

```plaintext
def emoji = overlay-align("center", "center", face, head)
def 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 lecture incorporates material from:

Kathi Fisler, Brown University
Jason Waterman, Vassar College