Where are we?
We've been using Pyret to write expressions that use:

- **numbers** (0, −10, 0.4),
- **strings** ("", "hi", "111"), and
- **images** (circle(2, "solid", "red")).

Which we modify or combine using operators or functions like +, **string-append**, and **above**.
Distinguishing types of data helps to catch mistakes.

If you try to give

a string to + or

a number to overlay,

we want Pyret to catch the problem right early, giving a helpful error message.
We’ve seen that we can create more complicated programs by composing function calls, e.g.,

\[ 1 + \left( \frac{2}{3} \right) \]

or

string-append("hello ",
            string-append("Pyret ", "world!"))
And we can give a name to the result of an expression, e.g.,

\[ total = 2 + 3 \]
Recap: Definitions
General form for a definition:

\[ \langle \text{name} \rangle = \langle \text{expression} \rangle \]
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.
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

\[
\text{width} = 400 \\
\text{height} = 600
\]

Now if we write

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

it gets evaluated:

\[
\begin{align*}
\rightarrow & \quad 400 \times \text{height} \\
\rightarrow & \quad 400 \times 600 \\
\rightarrow & \quad 240000
\end{align*}
\]
Example

Mary Berry needs to know how many cakes to bake for her cake shop.

To avoid running out or having too many, she likes to bake two cakes more than the number she sold the previous day.

E.g., if Mary sells eight cakes on Monday, she makes ten cakes on Tuesday.

Let's write some code to help Mary.
Mary sold eight cakes on Monday and wants to calculate how many to make on Tuesday:

\[ \text{num\text{-}sold} = 8 \]

\[ \text{num\text{-}sold} + 2 \]

On run, this will return 10
Mary sold eight cakes on Monday and wants to calculate how many to make on Tuesday:

\[
\text{num-sold} = 8
\]

\[
\text{num-sold} + 2
\]

On run, this will return 10

A definition names a value for later use in a computation. Essentially, we are assigning a label to a value that makes it easier to reuse the value in a program.
Mary sold eight cakes on Monday and wants to calculate how many to make on Tuesday:

\[
\text{num\text{-}sold} = 8
\]

\[
\text{num\text{-}sold} + 2
\]

On run, this will return 10

A \textit{definition} names a value for later use in a computation. Essentially, we are assigning a label to a value that makes it easier to reuse the value in a program.

An \textit{expression} performs computation without changing the information that Pyret maintains for running future expressions and return values.
Mary sold eight cakes on Monday and wants to calculate how many to make on Tuesday:

\[ \text{num-sold} = 10 \]

\[ \text{num-sold} + 2 \]

On run, this will return \textbf{12}

Once we make a definition, we can use it in subsequent computations.

We can manually change the value in the definition to change the final computation.
Defining functions
Remember functions from middle-school math:

Given $f(x) = 2 \cdot x$

$f(2) = 2 \cdot 2 = 4$

$f(6) = 2 \cdot 6 = 12$

Parameter stands for varying value
To form a function definition:

```plaintext
fun ⟨function-name⟩ (⟨arg-name⟩, ...):
 ⟨expression⟩
end
```
fun cakes(num-sold):
    num-sold + 2
end
fun cakes(num-sold):
    num-sold + 2
end
fun cakes(num-sold):
    num-sold + 2
end
fun **cakes**(num-sold):
    num-sold + 2
end
fun cakes(num-sold):
    num-sold + 2
end
fun cakes(num-sold):
  num-sold + 2
end
fun cakes(num-sold):
    num-sold + 2
end
fun cakes(num-sold):
    num-sold + 2
end

transform the data
fun cakes(num-sold):
    num-sold + 2
end
fun cakes(num-sold):
    num-sold + 2
end

special word to signal the function definition is done
Evaluation is the same as in algebra

\[
\text{fun } f(x): \text{num-cos}(x) + 2 \text{ end}
\]

\[
f(0) \rightarrow \text{num-cos}(0) + 2 \rightarrow 1 + 2 \rightarrow 3
\]
# Draw a traffic light

above(circle(40, "solid", "red"),
     above(circle(40, "solid", "yellow"),
           circle(40, "solid", "green")))
Draw a traffic light

above( circle(40, "solid", "red"),
above(circle(40, "solid", "yellow"),
circle(40, "solid", "green"))
# Draw a traffic light

above(circle(40, "solid", "red"),
above(circle(40, "solid", "yellow"),
circle(40, "solid", "green")))
# Draw a traffic light
above(circle(40, "solid", "red"),
    above(circle(40, "solid", "yellow"),
        circle(40, "solid", "green")))

# Can be changed to
fun bulb(color):
    circle(40, "solid", color)
end

above(bulb("red"),
    above(bulb("yellow"),
        bulb("green")))
fun bulb(color):
    circle(40, "solid", color)
end

fun traffic-light():
    above(bulb("red"),
         above(bulb("yellow"),
               bulb("green")))
end
We started by talking about *computation*, e.g., how do we interpret the expression

\[ 1 + (2 \times 3) \]
We started by talking about *computation*, e.g., how do we interpret the expression

\[
1 + (2 \times 3) \rightarrow 1 + 6
\]
We started by talking about *computation*, e.g., how do we interpret the expression

\[
1 + (2 \times 3) \rightarrow 1 + 6 \rightarrow 7
\]
We started by talking about *computation*, e.g., how do we interpret the expression

\[
1 + (2 \times 3) \\
\rightarrow 1 + 6 \\
\rightarrow 7
\]

But as we start writing function definitions, we need to think about *programming*.
Example
For Mary's cake shop, we want to determine the price of each cake based on the cost of the ingredients and the time to prepare it. The price is twice the cost of the ingredients plus 1/4 of the preparation time in minutes.
For Mary's cake shop, we want to determine the price of each cake based on the cost of the ingredients and the time to prepare it.

The price is twice the cost of the ingredients plus 1/4 of the preparation time in minutes.

Chocolate cake
Ingredients: $10
Preparation time: 20 minutes
For Mary's cake shop, we want to determine the price of each cake based on the cost of the ingredients and the time to prepare it.

The price is twice the cost of the ingredients plus 1/4 of the preparation time in minutes.

**Chocolate cake**
- Ingredients: $10
- Preparation time: 20 minutes

\[
\text{choc\text{-}cake\text{-}price} = (2 \times 10) + (0.25 \times 20)
\]
For Mary's cake shop, we want to determine the price of each cake based on the cost of the ingredients and the time to prepare it.

The price is twice the cost of the ingredients plus 1/4 of the preparation time in minutes.

- **Chocolate cake**
  - Ingredients: $10
  - Preparation time: 20 minutes
  
  \[
  \text{choc-cake-price} = (2 \times 10) + (0.25 \times 20)
  \]

- **Cheesecake**
  - Ingredients: $15
  - Preparation time: 36 minutes
For Mary's cake shop, we want to determine the price of each cake based on the cost of the ingredients and the time to prepare it.

The price is twice the cost of the ingredients plus 1/4 of the preparation time in minutes.

**Chocolate cake**
- Ingredients: $10
- Preparation time: 20 minutes

**Cheesecake**
- Ingredients: $15
- Preparation time: 36 minutes

\[
\text{choc-cake-price} = (2 \times 10) + (0.25 \times 20)
\]

\[
\text{cheesecake-price} = (2 \times 15) + (0.25 \times 36)
\]
We use functions to avoid repetitive code when we need to perform the same operations on different values.

\[
\text{choc-cake-price} = (2 \times 10) + (0.25 \times 20)
\]

\[
\text{cheesecake-price} = (2 \times 15) + (0.25 \times 36)
\]
We use functions to avoid repetitive code when we need to perform the same operations on different values.

\[ \text{choc-cake-price} = (2 \times 10) + (0.25 \times 20) \]

\[ \text{cheesecake-price} = (2 \times 15) + (0.25 \times 36) \]
We use functions to avoid repetitive code when we need to perform the same operations on different values.

\[ \text{choc-cake-price} = (2 \times 10) + (0.25 \times 20) \]

\[ \text{cheesecake-price} = (2 \times 15) + (0.25 \times 36) \]
fun cake-price(ingredients-cost, prep-time):
    (2 * ingredients-cost) + (0.25 * prep-time)
end
fun cake-price(ingredients-cost, prep-time):
    (2 * ingredients-cost) + (0.25 * prep-time)
end

The parameters are the values passed into the function that it needs to know for each operation.
fun cake-price(ingredients-cost, prep-time):
  (2 * ingredients-cost) + (0.25 * prep-time)
end

Expression repeated each time the function is called
fun cake-price(ingredients-cost, prep-time):
   (2 * ingredients-cost) + (0.25 * prep-time)
end

To calculate the price of chocolate cake or cheesecake, you simply call your function and pass in the relevant values:

# Price of chocolate cake
cake-price(10, 20)

# Price of cheesecake
cake-price(15, 36)
Type annotations
fun cake-price (ingredients-cost :: Number, prep-time :: Number):
  (2 * ingredients-cost) + (0.25 * prep-time)
end

We specify the type of each parameter so that Pyret will check that we pass in the right kind of values, just like for built-in operations like + and above.
fun cake-price(ingredients-cost :: Number, prep-time :: Number) -> Number:
    (2 * ingredients-cost) + (0.25 * prep-time)
end

And we can specify the type of value the function returns.
fun cake-price (ingredients-cost :: Number, prep-time :: Number) -> Number:
    doc: "Calculate price of cake based on ingredient cost and prep time"
    (2 * ingredients-cost) + (0.25 * prep-time)
end

A docstring explains what the function does.
“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
fun cakes(num-sold):
    num-sold + 2
end
fun cakes(num-sold :: Number):
  num-sold + 2
end
fun cakes(num-sold :: Number) -> Number:
    num-sold + 2
end
fun cakes(num-sold :: Number) -> Number:
  doc: "Compute the number of cakes to make based on the previous number sold"
  num-sold + 2
end
fun sqr(dimen):
    rectangle(dimen, dimen, "solid", "blue")
end
fun \texttt{sqre}(\texttt{dimen :: Number}) \rightarrow \texttt{Image}:
  \texttt{doc}: "Make a solid blue square of the given side length"
  \texttt{rectangle}(\texttt{dimen, dimen, "solid", "blue"})
end
fun rectangle-area(r):
    image-height(r) * image-width(r)
end
fun rectangle-area(r :: Image) -> Number:
  doc: "Return the rectangular area of the image"
  image-height(r) * image-width(r)
end
fun dupe-string(str):
    string-append(str, str)
end
fun dupe-string(str :: String) -> String:
    doc: "Return a string containing the input repeated"
    string-append(str, str)
end
Testing functions & why we care
The Jane Austen Collection: Deluxe 6-Volume Slipcase Edition (Arcturus Collector’s Classics)
by Jane Austen | Sep 19, 2016
Stars: 4.5 out of 5 stars | 274 reviews
Hardcover
$15.99

The Jane Austen Society: A Novel
by Natalie Jenner | May 26, 2020
Stars: 4.5 out of 5 stars | 7 reviews
Hardcover
$16.99

Jane Was Here: An Illustrated Guide to Jane Austen's England
by Nicola Jacobsen, Diane MacLennan, et al. | Jun 8, 2020
Stars: 3.5 out of 5 stars | 3 reviews
Hardcover
$15.99
The Jane Austen Collection: Deluxe 6-Volume Slipcase Edition (Arcturus Collector's Classics)
by Jane Austen | Sep 15, 2016

5.0 out of 5 stars - 274

Hardcover

$19.99

Prime Get it as soon as Wed, Sep 9
FREE Shipping by Amazon

More Buying Choices
$17.60 ($2 used & new offers)

Kindle

10% off - $17.91
Available instantly

Paperback

Other format: Audible CD

The Jane Austen Society: A Novel
by Natalie Jenner | May 28, 2020

4.3 out of 5 stars - 711

Hardcover

$16.99

Get it as soon as Wed, Sep 9
FREE Shipping on your first order shipped by Amazon

More Buying Choices
$15.68 ($4 used & new offers)

Kindle

$1.99

Audible Audiobook

$16.99

Free with Audible trial

Other formats: Audible CD, Paperback

Jane Was Here: An Illustrated Guide to Jane Austen's England
by Nicole Jacobson, Devon Maclellan, et al. | Jun 8, 2020

3.5 out of 5 stars - 3

Hardcover

$15.00

Get it as soon as Wed, Sep 9
FREE Shipping on qualified orders over $25.
fun cakes(num-sold :: Number) -> Number:
  doc: "Compute the number of cakes to make based on
  the previous number sold"
  num-sold + 2
end
fun cakes(num-sold :: Number) -> Number:
  doc: "Compute the number of cakes to make based on the previous number sold"
  num-sold + 2
where:
  cakes(0) is 2
  cakes(107) is 109
end
Booleans and if expressions
true
false
>>> true and false
false
>>> true or False
true
>>> (1 < 2) and (2 > 3)
false
>>> (1 <= 0) or (1 == 1)
true
To combine Boolean values, we can use \textbf{and}:

\[
\langle expr1 \rangle \text{ and } \langle expr2 \rangle
\]

and \textbf{or}:

\[
\langle expr1 \rangle \text{ or } \langle expr2 \rangle
\]

Evaluation of \textbf{and} stops – is “short-circuited” – as soon as an expression evaluates to false.

Evaluation of \textbf{or} stops as soon as an expression evaluates to true.
To change an expression that evaluates to true to be false or vice versa, use `not`.

```python
>>> not(1 == 0)
true
```
\[ I_1 = \text{rectangle}(10, 20, \text{"solid"}, \text{"red"}) \]
\[ I_2 = \text{rectangle}(20, 10, \text{"solid"}, \text{"blue"}) \]

\[ \text{image-width}(I_1) < \text{image-width}(I_2) \]
rect = rectangle(10, 20, "solid", "red")

if image-width(rect) < image-height(rect):
    "tall"
else:
    "wide"
end
To form an **if** expression:

```
if ⟨expression⟩:
  ⟨expression⟩
else:
  ⟨expression⟩
end
```
Evaluation rule for if expressions

1. If the question expression is not a value, evaluate it, and replace with value
2. If the question is true, replace entire if expression with true answer expression
3. If the question is false, replace entire if expression with false answer expression
4. The question is a value other than true or false, so produce an error
rect = rectangle(10, 20, "solid", "red")

if image-width(rect) < image-height(rect):
    "tall"
else:
    "wide"
end

What if, instead of producing a Boolean to say if an image is tall or not, we classify them as “tall”, “square”, or “wide”?
rect = rectangle(10, 20, "solid", "red")

if image-width(rect) < image-height(rect):
    "tall"
else if image-width(rect) == image-height(rect):
    "square"
else:
    "wide"
end
Functions with conditionals
fun image-type(img :: Image) -> String:
  doc: "Classify an image as tall, square, or wide"
  if image-width(img) < image-height(img):
    "tall"
  else if image-width(img) == image-height(img):
    "square"
  else:
    "wide"
end

where:
  image-type(rect) is "tall"
end
```plaintext
rect = rectangle(10, 20, "solid", "red")

fun image-type(img :: Image) -> String:
    doc: "Classify an image as tall, square, or wide"
    if image-width(img) < image-height(img):
        "tall"
    else if image-width(img) == image-height(img):
        "square"
    else:
        "wide"
end

where:
    image-type(rect) is "tall"
    image-type(rectangle(10, 10, "solid", "blue")) is "square"
    image-type(rectangle(20, 10, "solid", "blue")) is "wide"
end
```
Acknowledgments

This lecture incorporates material from:

Kathi Fisler, Brown University
Gregor Kiczales, University of British Columbia