Evaluating Functions and Conditionals

7 September 2022
Reminder: Labs need to be checked off before you submit on Gradescope

- Coaches or I can check them during the lab period
- If you need more time, go to coaching hours during the week
I will make every effort to give each of you the attention and feedback you need to be successful in this course – but there’s only one of me! Therefore, I rely on the coaches to help me help answer your questions.

In addition to working during our labs each week, each coach will be available to help you in the Agile Lab (sc 006) at scheduled times.

**Important:** The coaches are prohibited from giving you the solutions to labs and assignments, but they are able to guide you as you work to solve your programming tasks. When this works well, they will help you answer your own questions!
Assignment 1 due midnight on Wednesday
Assignment 2 comes out on Thursday
Linux tutorial

David Frey, CS Department System Administrator

Wednesday, 14 September   3 p.m.   Sanders Physics 309
Linux tutorial

David Frey, CS Department System Administrator

Wednesday, 14 September  3 p.m.  Sanders Physics 309

Not Sanders Classroom!
Where are we?
We’ve been using Pyret to write expressions that use:

Data, including

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

Which we modify or combine using operators like + and * and 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 away, giving a helpful error message.
Failing fast is a nonintuitive technique. “Failing immediately and visibly” sounds like it would make your software more fragile, but it actually makes it more robust. Bugs are easier to find and fix, so fewer go into production. A default value, everything will seem fine.

https://www.martinfowler.com/leef/Sofware/fallFast.html

Fail Fast - Martin Fowler

People also ask:

- What is fail fast concept?
- What is a fail fast culture?
- Why is Agile fail fast?
- What is fail fast innovation?

https://en.wikipedia.org/wiki/Fail-fast

Fail-fast - Wikipedia

In systems design, a fail-fast system is one which immediately reports at its interface any condition that is likely to indicate a failure.
We’ve seen that we can create more complicated programs by composing function calls, e.g.,

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

or

\[
\text{string-append}("hello ", \\
\quad \text{string-append}("Pyret ", "world!"))
\]
And we can give a name to the result of an expression, e.g.,

\[ total = 2 + 3 \]
And we can give a name to the result of an expression, e.g.,

\[ total = 2 + 3 \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td></td>
</tr>
</tbody>
</table>
And we can give a name to the result of an expression, e.g.,

\[ \text{\emph{total}} = 2 + 3 \]
\[ \rightarrow \text{\emph{total}} = 5 \]
And we can give a name to the result of an expression, e.g.,

\[
\text{total} = 2 + 3 \\
\rightarrow \text{total} = 5
\]
And we can give a name to the result of an expression, e.g.,

\[
\begin{align*}
\text{total} & = 2 + 3 \\
\rightarrow \text{total} & = 5 \\
\text{new-total} & = \text{total} + 1
\end{align*}
\]

Directory

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>5</td>
</tr>
</tbody>
</table>
And we can give a name to the result of an expression, e.g.,

\[
\begin{align*}
\text{total} &= 2 + 3 \\
&
\to \text{total} = 5
\end{align*}
\]

\[
\begin{align*}
\text{new-total} &= \text{total} + 1
\end{align*}
\]
And we can give a name to the result of an expression, e.g.,

\[
\text{total} = 2 + 3 \\
\rightarrow \text{total} = 5
\]

\[
\text{new-total} = \text{total} + 1
\]
And we can give a name to the result of an expression, e.g.,

\[
\begin{align*}
total &= 2 + 3 \\
    &\rightarrow total = 5
\end{align*}
\]

\[
\begin{align*}
new\text{-}total &= total + 1 \\
    &\rightarrow new\text{-}total = 5 + 1
\end{align*}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>5</td>
</tr>
<tr>
<td>new-total</td>
<td></td>
</tr>
</tbody>
</table>
And we can give a name to the result of an expression, e.g.,

\[
\begin{align*}
total &= 2 + 3 \\
\rightarrow total &= 5 \\
new-total &= total + 1 \\
\rightarrow new-total &= 5 + 1 \\
\rightarrow new-total &= 6
\end{align*}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>5</td>
</tr>
<tr>
<td>new-total</td>
<td></td>
</tr>
</tbody>
</table>
And we can give a name to the result of an expression, e.g.,

\[ total = 2 + 3 \]
\[ \rightarrow total = 5 \]

\[ new-total = total + 1 \]
\[ \rightarrow new-total = 5 + 1 \]
\[ \rightarrow new-total = 6 \]
Defining functions
Remember functions from middle-school math:

Given $f(x) = \cos(x) + 2$

$f(0) = 1 + 2 = 3$

Parameter stands for varying value
Pyret functions work the same way:

```pyret
fun f(x): num-cos(x) + 2 end
```

\[
\begin{align*}
f(0) & \rightarrow \text{num-cos}(0) + 2 \\
& \rightarrow 1 + 2 \\
& \rightarrow 3
\end{align*}
\]
Function definitions in Pyret have this form:

```pyret
fun ⟨function-name⟩(⟨parameter1⟩, ...) : ⟨expression⟩
end
```
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.
fun cakes-to-make(num-sold):
    num-sold + 2
end
fun cakes-to-make(num-sold):
    num-sold + 2
end

Keyword to define a function
fun cakes-to-make(num-sold):
    num-sold + 2
end
fun cakes-to-make(num-sold):
    num-sold + 2
end
fun cakes-to-make(num-sold):
    num-sold + 2
end
fun cakes-to-make(num-sold):
    num-sold + 2
end

Parameter
fun cakes-to-make(num-sold):
    num-sold + 2
end
fun cakes-to-make(num-sold):
    num-sold + 2
end

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

Keyword to signal the function definition is done
Functional abstraction
# 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

```
\[
\text{above(}
\quad \text{circle(40, "solid", "red"),}
\quad \text{above(}
\quad \quad \text{circle(40, "solid", "yellow"),}
\quad \quad \text{circle(40, "solid", "green"))}
\text{Unchanging}
\text{Varying}
\]
```

![Traffic light image]
# 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
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

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

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

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

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

\[
cheesecake\text{-}price = (2 \times 15) + (0.25 \times 36)
\]
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.

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

\[
(cheesecake\text{-price}) = (2 \times 15) + (0.25 \times 36)
\]
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

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

# Price of cheesecake
cake-price(15, 36)

To calculate the price of chocolate cake or cheesecake, you just call your function and pass in the relevant values!
Improving our function definitions
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:
  (2 * ingredients-cost) + (0.25 * prep-time)
end

>> cake-price(2, 3)
4.75

>> cake-price("banana", "bundt")

The Number annotation

was not satisfied by the value

"banana"

(Show program evaluation trace...)

>>
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

Additionally, a docstring explains what the function does.
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)
where:
  # Price of chocolate cake
  cake-price(10, 20) is (2 * 10) + (0.25 * 20)
  # Price of cheesecake
  cake-price(15, 36) is (2 * 15) + (0.25 * 36)
end
use context essentials2021

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)
where:
  # Price of chocolate cake
cake-price(10, 20) is (2 * 10) + (0.25 * 20)
  # Price of cheesecake
  cake-price(15, 36) is (2 * 15) + (0.25 * 36)
end
use context essentials2021

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) * (0.3 * prep-time)
  where:
  # Price of chocolate cake
cake-price(10, 20) is (2 * 10) * (0.25 * 20)
  # Price of cheesecake
cake-price(15, 36) is (2 * 15) * (0.25 * 36)
end

0 TESTS PASSED  2 TESTS FAILED

Test 1: Failed
The test operator is failed for the test:

cake-price(10, 20) is (2 * 10) * (0.25 * 20)

It succeeds only if the left side and right side are equal.
The left side was:
26
The right side was:
25

Test 2: Failed
The test operator is failed for the test:

cake-price(15, 36) is (2 * 15) * (0.25 * 36)

It succeeds only if the left side and right side are equal.
The Jane Austen Collection: Deluxe 6-Volume Slipcase Edition (Arcturus Collector's Classics)
by Jane Austen | Sep 15, 2016

Hardcover
$19.99

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

More Buying Choices
$17.60 (12 used & new offers)

Kindle
$10.99

Available instantly

Paperback
Other format: Audible Audio Edition

The Jane Austen Society: A Novel
by Natalie Jenner | May 26, 2020

Hardcover
$16.99

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

More Buying Choices
$13.68 (4 used & new offers)

Kindle
$13.99

Available instantly

Audible Audiobook
$10.99

Free with Audible trial

Other formats: Audible Audio Edition, Paperback

Jane Was Here: An Illustrated Guide to Jane Austen's England
by Nicola Jacobson, Devon McLeamy, et al. | Jun 3, 2020

Hardcover
$15.99

Get it as soon as Wed, Sep 9
FREE Shipping on qualified orders over $25 by Amazon
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)
where:
  rectangle-area(rectangle(0, 0, "solid", "black"))
  is 0
  rectangle-area(rectangle(2, 3, "outline", "blue"))
  is 6
end
Booleans and if expressions
true
false
To combine Boolean values, we can use **and**:

\[
\langle \text{expression 1} \rangle \text{ and } \langle \text{expression 2} \rangle
\]

and **or**:

\[
\langle \text{expression 1} \rangle \text{ or } \langle \text{expression 2} \rangle
\]

Evaluation of **and** stops – is “short-circuited” – as soon as one of the expressions being combined evaluates to **false**.

Evaluation of **or** stops as soon as one of the expressions evaluates to **true**.
>>> true and false
false
>>> true or false
true
>>> (1 < 2) and (2 > 3)
false
>>> (1 <= 0) or (1 == 1)
true
To change an expression that evaluates to `true` to be `false` or vice versa, use `not`:

```python
>>> not(true)
false
>>> not(1 == 0)
true
```
$i_1 = \text{rectangle}(10, 20, \"solid\", \"red\")$

$i_2 = \text{rectangle}(20, 10, \"solid\", \"blue\")$

\(\text{image-width}(i_1) < \text{image-width}(i_2)\)
rect = rectangle(10, 20, "solid", "red")

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

```
if ⟨expression⟩:
  ⟨expression⟩
else:
  ⟨expression⟩
end
```

- True–false question
- True (“then”) answer
- False (“else”) answer
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):
    "portrait"
else:
    "landscape"
end

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

if image-width(rect) < image-height(rect):
   "portrait"
else if image-width(rect) == image-height(rect):
   "square"
else:
   "landscape"
end
rect = rectangle(10, 20, "solid", "red")

fun image-type(img :: Image) -> String:
    doc: "Classify an image as portrait, square, or landscape"
    if image-width(img) < image-height(img):
        "portrait"
    else if image-width(img) == image-height(img):
        "square"
    else:
        "landscape"
    end
where:
    image-type(rect) is "portrait"
end
rect = rectangle(10, 20, "solid", "red")

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

where:
  image-type(rect) is "portrait"
  image-type(rectangle(10, 10, "solid", "blue")) is "square"
  image-type(rectangle(20, 10, "solid", "blue")) is "landscape"
end
Acknowledgments

This class incorporates material from:

Kathi Fisler, Brown University
Gregor Kiczales, University of British Columbia
Peter Lemieszewski, Vassar College