Evaluating Functions and Conditionals

7 September 2022
Assignment 1 due 11:59pm on Wednesday

Assignment 2 comes out on Thursday
Where are we?
We've been using Pyret to write expressions that use:

Data, including

- **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.,

\[ 1 + (2 / 3) \]

or

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

\[ total = 2 + 3 \]
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:

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

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

```python
fun ⟨function-name⟩ (⟨arg-name⟩, ...):
  ⟨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

`special word 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

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

special word to signal the function definition is done
Functional abstraction
# Draw a traffic light

```latex
\text{above( circle(40, "solid", "red"), }
\text{above(circle(40, "solid", "yellow"), }
\text{circle(40, "solid", "green")))}
```

Unchanging

Varying
# 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.

**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.

\[
(2 \times \text{ingredients-cost}) + (0.25 \times \text{prep-time})
\]

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

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

*Parameters*

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)
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.
```plaintext
use context essentials2021

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) \rightarrow Number:
    doc: "Calculate price of cake based on ingredient cost and prep time"
    (2 \times ingredients-cost) + (0.25 \times prep-time)
end

Additionally, 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-to-make(num-sold :: Number) -> Number:
  doc: "Compute the number of cakes to make based on the previous number sold"
  num-sold + 2
end
fun cakes-to-make(num-sold :: Number) -> Number:
    doc: "Compute the number of cakes to make based on the previous number sold"
    num-sold + 2
where:
    cakes-to-make(0) is 2
    cakes-to-make(107) is 109
end
fun cakes-to-make(num-sold :: Number) -> Number:
    doc: "Compute the number of cakes to make based on the previous number sold"
    num-sold + 3
    where:
    cakes-to-make(0) is 2
    cakes-to-make(107) is 109
end

cakes-to-make
0 out of 2 tests passed in this block.

Test 1: Failed

The test operator is failed for the test:
7 cakes-to-make(0) is 2
It succeeds only if the left side and right side are equal.
The left side was:
3
The right side was:
2

Test 2: Failed

The test operator is failed for the test:
8 cakes-to-make(107) is 109
It succeeds only if the left side and right side are equal.
The left side:
The Jane Austen Collection: Deluxe 6-Volume Slipcase Edition (Arcturus Collector's Classics)

by Jane Austen | Sep 15, 2016

Hardcover

$24.00

Pre-order 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

$14.99

Get it as soon as Wed, Sep 9
FREE Shipping on your first order shipped by Amazon
More Buying Choices
$11.98 (44 used & new offers)

Kindle

$11.99

Available instantly

Audible Audiobook

$10.99

Free with Audible trial

Other formats: Audible Audiobook, Paperback

Jane Was Here: An Illustrated Guide to Jane Austen's England

by Nicola Jacobsen, Devon MacLaren, et al. | Jun 9, 2020

Hardcover

$32.00

Get it as soon as Wed, Sep 9
FREE Shipping on eligible orders over $25 shipped 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
Booleans and \texttt{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(1 == 0)
true
```
i1 = rectangle(10, 20, "solid", "red")
i2 = rectangle(20, 10, "solid", "blue")

image-width(i1) < image-width(i2)
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
```

- **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. If 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
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"
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"
Acknowledgments

This class incorporates material from:

Kathi Fisler, Brown University
Gregor Kiczales, University of British Columbia
Jonathan Gordon, Vassar College