Functions... kinda like $f(x)$
A quick review of last week’s concepts

• We’ve been using Pyret to write expressions that use:
  1. Data, including numbers (0, -10, 0.4),
  2. strings ("", "hi", "111"),
  3. images (circle(2, "solid", "red")).

• Which we modify or combine using operators or functions
  1. +
  2. string-append
  3. overlay
A quick review of last week’s concepts

• We’ve been using Pyret to write expressions that use:
  1. Data, including numbers (0, -10, 0.4),
  2. strings ("", "hi", "111"),
  3. images (circle(2, "solid", "red")).

• Which we modify or combine using operators or functions
  1. + Operator
  2. string-append Function
  3. overlay Function
Errors

• IRL Software Problems are expensive when found by customers
  • In terms of cost of lost revenue, fixing the problem, brand satisfaction
  • Typically called “run time errors”
  • Less expensive if found before released to customers

• The cost of a problem is lower… the earlier a problem is discovered

• Lowest cost: finding a problem immediately after we write it!
  • This is what pyret does as you type code
  • This is also what happens when you click on Run

• In Agile programming terms: Fail Fast
Fail Fast

• ...google search:

https://www.agile-academy.com › ... › Fail Fast

Fail Fast | Agile Dictionary

People also ask

Why is agile fail fast?

What is fail fast in programming?

Essentially, fail fast (a.k.a. fail early) is to code your software such that, when there is a problem, the software fails as soon as and as visibly as possible, rather than trying to proceed in a possibly unstable state.

https://stackoverflow.com › questions › what-does-the-ex...

What does the expression "Fail Early" mean, and when would you want to ...

Search for: What is fail fast in programming?
What else did we observe last week?

• We can create (more) sophisticated code by combining functions and operators

• ...essentially creating expressions
  • 1 + (7 / 8)
  • string-append(“Computer”, “Science”)

• We can name our expressions too

• ...more precisely, the results of our expressions
  • my-major = string-append(“Computer”, “Science”)

8/30/2022

CMPU 101: Problem Solving and Abstraction
Today’s Topic: Functions

• In mathematics:
  • \( f(x) = x^3 + 2x + 1 \)

• In Pyret Programming:
  • We need a way to tell Pyret we have a function:
    
    ```pyret
    fun f(x): = (x * x * x) + (2 * x) + 1
    end
    ```
Pyret Function Syntax

Function definitions in Pyret have this form:

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

- Angle brackets `< >` refer to something that is optional
  - Technically, the ellipse should have angle brackets too!
- Another name for `arg-name` is parameter
SSE: Super Simple 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.
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
fun cakes-to-make(num-sold):
  num-sold + 2
end

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

custom word to signal
the function definition
is done
# Draw a traffic light

```python
above(circle(40, "solid", "red"), above(circle(40, "solid", "yellow"), circle(40, "solid", "green")))
```

Unchanging

Varying
Functional Abstraction 2/3

# 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")))
Functional Abstraction: 3/3 (kicking it up a notch)

fun bulb(color):
    circle(40, "solid", color)
end

fun traffic-light():
    above(bulb("red"),
        above(bulb("yellow"),
            bulb("green")))
end
Consider Mary’s cake shop (again)

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.

One approach: consider each cake, separately

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

\[
\text{cheesecake-price} = (2 \times 15) + (0.25 \times 36)
\]
Functional Abstraction: Ace of Cakes Functions

• Looking more closely...
  • The price is twice the cost of the ingredients plus 1/4 of the preparation time in minutes.
• Use functions to avoid repetitive code when performing the same operations with different values.
• Purple font/arrows: different values? Hmm... should be a parameter!

- **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

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

• Parameters: generic names representing values that are passed into a function & are required/needed for creating a result.

```plaintext
fun cake-price(ingredients-cost, prep-time):
    (2 * ingredients-cost) + (0.25 * prep-time)
end
```

• Using our cake example, we can now calculate cost of any kind of cake by calling the cake-price function.

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

# Price of cheesecake
cake-price(15, 36)
```
Making bitter-batter better with butter functions better

- Improved definition: Parameters: generic names representing values of a particular type that are passed into a function & are required/needed for creating a result.

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 the right type of values are actually being passed. Why does this matter?
Making bitter-batter-better with butter functions better

• Improved definition: Parameters: generic names representing values of a particular type that are passed into a function & are required/needed for creating a result.

```python
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 the right type of values are actually being passed. Why does this matter?

  Answer: so that we can fail fast and discover problems faster than if we didn’t check
Making functions better (returning a result)

• We can do this for the function result (i.e. return type) too!

```haskell
fun
  cake-price(ingredients-cost :: Number, prep-time :: Number) -> Number:
    (2 * ingredients-cost) + (0.25 * prep-time)
end
```

• We specify the type of the return value to maintain data consistency
  i.e. so that we can fail fast when calling a function and naming the result!
Using pyret

```pyret
use context essentials2021

fun cake-price(ingredients-cost :: Number, prep-time :: Number) -> Number:
  (2 * ingredients-cost) + (0.25 * prep-time)
end

>> cake-price(5, 9)
12.25

>> cake-price("bam", "chocolate")

The Number annotation

was not satisfied by the value
"bam"
(Show program evaluation trace...)

>> cost = cake-price(5, 9)
```

CMPU 101: Problem Solving and Abstraction

8/30/2022
Making functions better (epilogue)

• It’s a good idea to let the instructor other programmers know what a 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)
end

• We document the function so that a user of the function, or one who maintains it, can understand and use it properly.
  i.e. so that we don’t have to fail at all!
More On Failing Fast (i.e. testing)

• Consider the following function which includes some test information

```pyret
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
```

• But what if we happen to make a typo in pyret...
More On Failing Fast (i.e. testing)

```haskell
use context essentials

fun cakes-to-make(num-sold :: Number) -> Number: doc "Compute the number of cakes to make based on the previous number sold"
num-sold + 22

where:
cakes-to-make(0) is 2

end
```

Tests:

<table>
<thead>
<tr>
<th>Test</th>
<th>Passed</th>
<th>Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 out of 2 tests passed in this block.

Test 1: Failed

The test operator is failed for the test:
```
cakes-to-make(0) is 2
```

It succeeds only if the left side and right side are equal.

The left side was:
22

The right side was:
2

Test 2: Failed

The test operator is failed for the test:
```
```
• Consider the following function which uses an image!

```haskell
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
```
Another Testing Example: epilogue

```
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
```
Another Testing Example: epilogue

Achievement unlocked: pyret talks like a pirate!

```
use context essentials2021
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

Looks shipshape, both tests passed, mate!
rectangle-area
All 2 tests in this block passed.
```