**3.1例子:月球重量**

假设我们负责给一组宇航员配备探月设备。 我们必须确定它们在月球表面的重量。 我们知道如何做到这一点——我们在前面看到过这个表达式[REF]——但是一遍又一遍地写它是很无聊的。 此外，如果我们复制或重新键入一个表达式多次，迟早我们一定会犯转录错误。 这是干法原理的一个实例。 另外，纠正错误本身就是一个有趣的计算机科学主题，我们将在稍后讨论[参考文献]。 (本段包含链接：[DRY principle](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself" \t "_blank)  )

当我们计算月球的重量时

100 \* 1/6

150 \* 1/6

90 \* 1/6

-我们看到有些部分是“固定的”，有些部分是“变化的”。 固定的部分是我们不想重复的; 变化的部分是我们没有选择的(并且希望有变化的自由)。 如果能有一个软件包能清楚地说明这一点，那就太好了。

我们要做的是写一个函数。 函数接受一个或多个参数，这些参数是变化的部分。 具体来说，我们创建函数的方法是

•  写下一些期望计算的例子。

•  识别哪些部分是固定的(上面，\* 1/6)，哪些部分在变化(上面，100,150,90…)。

•  对于每个变化的部分，给它一个名称(比如地球重量)，这将是代表它的参数。

•  重写例子，使之与这个参数有关:

earth-weight \* 1/6

***现在做!***

为什么只有一个表达方式，而之前我们有很多?

我们只有一个表达式，因为关键是去掉所有变化的部分，用参数替换它们。

•  将函数命名为一些暗示性的东西:例如，moon-weight。

•  编写表达式周围函数的语法:

•  **有趣的<函数名>(<参数>):**

•     表达式写在这里>

**结束**

其中表达式称为函数体。

哇，看起来工作量很大啊! 但最终的结果非常简单:

**乐趣moon-weight (earth-weight):**

   earth-weight \* 1/6

**结束**

我们会一遍又一遍地重复同样的步骤，最终它们会变得如此直观，以至于我们甚至不记得我们实际上是采取步骤从例子到函数的:它会变成一个单一的，自然的步骤。

我们如何使用它? 从Pyret的观点来看，moon-weight只是另一个操作符，就像number -expt或overlay一样。 因此:

moon-weight (100)

moon-weight (150)

moon-weight (90)

将产生与我们开始时相同的答案，但我们不会在公式中由于复制或重新键入而犯任何错误。

**3.2例子:日本国旗**

让我们创建另一个函数。 还记得我们的日本国旗吗? 每次我们想要一个不同大小的标志，我们必须改变单位的值，并重新运行整个程序。 相反，我们应该创建一个生成日本国旗的函数。

这个函数需要多少参数? 回到前面的代码，我们看到唯一真正改变的是单元。 其他的都是从这里计算出来的。 因此，我们应该将unit转换为参数，并保持其余的计算(已经是单位形式)不变:

**有趣的日本国旗(单位):**

   bg-width =单元\* 3

   bg-height =单位\* 2

   circ-rad = 3/5 \* 1/2 \* bg-height

   red-circ =圆圈(circ-rad， "solid"， "red")

   白色矩形=矩形(bg-width, bg-height， "solid"， "white")

   覆盖(red-circ white-rect)

**结束**

这个函数体创建几个局部[REF]变量，并最终生成覆盖表达式的结果，即标志形状。 因此，我们可以多次使用它:

日本国旗(100)

日本国旗(200)

日本国旗(50)

无需在更改之间重新运行程序。 注意，如果生成的图像很大，Pyret将用它的缩略图版本替换acutal图像。 单击缩略图查看完整图像。

**3.3测试:跟踪实例**

在上面的每一个函数中，我们都从一些我们想要计算的例子开始，从那里推广到一个通用公式，把它变成一个函数，然后用这个函数来代替原来的表达式。

现在我们已经完成了，最初的例子有什么用呢? 把它们扔掉似乎很有诱惑力。 然而，有一条关于软件的重要规则您应该学习:软件的发展。 随着时间的推移，任何具有任何用途的程序都将发生变化和增长，最终可能产生与最初不同的值。 有时这些错误是故意的，但有时这些错误是错误的结果(包括一些愚蠢但不可避免的错误，如在键入时意外添加或删除文本)。 因此，保留这些示例以备将来参考总是很有用的，因此如果函数偏离了它应该泛化的示例，您可以立即得到警告。

Pyret使这很容易做到。 每个函数都可以附带一个where子句来记录示例。 例如，我们的月球重量可以修改为:

**乐趣moon-weight (earth-weight):**

   earth-weight \* 1/6

**地点:**

   月球重量(100)是100 \* 1/6

   月球重量(150)是150 \* 1/6

   月球重量(90)是90 \* 1/6

**结束**

当以这种方式编写时，Pyret将在每次运行程序时检查答案，并通知您是否将函数更改为与这些示例不一致。

***现在做!***

检查这个! 更改公式—例如，用。替换函数体

earth-weight \* 1/3

看看会发生什么。

当然，对于这样简单的函数，您不太可能犯错误(除了输入错误)。 毕竟，这些示例非常类似于函数本身的主体。 然而，稍后，我们将看到示例可能比主体简单得多，因此不再那么容易判断它们的行为是否相同，并且我们将发现，使主体与示例匹配是很困难的。 事实上，在实际的软件生产中，这是非常常见的，以至于专业的程序员总是写下这样的例子——称为测试——以确保他们的程序按照他们所期望的那样运行。

**3.4类型注解**

假设我们把月亮的重量记在一根绳子上:

moon-weight(阿姆斯壮)

***现在做!***

会发生什么呢?

Pyret生成一个错误，说您不能将一个数字乘以一个字符串(教您算术的人会很高兴听到这一点)。

在这么小的函数中，这几乎无关紧要。 但如果你有一个更大的功能，从它的内部深处得到类似的错误将是令人沮丧的。 更糟的是，如果您得到一个别人编写的函数，您需要读取整个函数(可能会更大)，以确定它使用和生成的值类型。

幸运的是，我们可以做得更好。 Pyret允许您在指示其值的函数上编写注释。 具体来说，在月球重量的情况下，因为它消耗和产生数字，我们会写:

**有趣的月球重量(地球重量::数字)->数字:**

   earth-weight \* 1/6

**结束**

为了简洁起见，我们省略了where示例，但是您也可以编写这些示例。 现在，仅仅通过阅读这个函数，您就可以知道它使用了一个数字(::number部分)，并且还生成了一个数字(-> number部分)。

***现在做!***

当你跑月球重量(“阿姆斯特朗”)会发生什么?

***现在做!***

日本国旗上会有什么注释?

因为japan-flag消耗一个数字并生成一个图像，所以我们写道:

**fun日本国旗(单位::号)->图像:**

   bg-width =单元\* 3

   bg-height =单位\* 2

   circ-rad = 3/5 \* 1/2 \* bg-height

   red-circ =圆圈(circ-rad， "solid"， "red")

   白色矩形=矩形(bg-width, bg-height， "solid"， "white")

   覆盖(red-circ white-rect)

**结束**

注意，这些注释显然是可选的:在本节之前，我们的函数都没有这些注释。 事实上，您可以在一个地方使用注释，而不是在另一个地方。 此外，您还可以在任何新变量上添加注释，而不仅仅是参数中的变量:例如，japan-flag中的变量也可以添加注释。

***现在做!***

在每个空格填上注释:

**fun日本国旗(单位::号)->图像:**

   宽度::\_ =单位\* 3

   bg-height:: \_ = unit \* 2

   circ-rad:: \_ = 3/5 \* 1/2 \* bg-height

   red-circ: \_\_\_ =圆(circ-rad“固体”,“红”)

   矩形(bg-width, bg-height， "solid"， "white")

   覆盖(red-circ white-rect)

**结束**

完整注释的函数是:

**fun日本国旗(单位::号)->图像:**

   宽度::Number =单位\* 3

   bg-height:: Number = unit \* 2

   circ-rad:数值= 3/5 \* 1/2 \* bg-height

   red-circ:: Image =圆圈(circ-rad，“solid”，“red”)

   白色矩形::Image =矩形(bg-width, bg-height， "solid"， "white")

   覆盖(red-circ white-rect)

**结束**

***现在做!***

更改其中一项注释为不正确:例如，

red-circ:: Number =圆圈(circ-rad， "solid"， "red")

•  什么时候会出错? 是当您单击Run时，还是仅当您实际使用japan-flag时?

•  错误指的是程序的哪一部分?

我们在注释中放入的东西——number、String等等——称为类型。 类型帮助我们区分不同类型的数据。 每个值都有一个类型，并且没有一个值具有多个类型。 因此，3是一个数字(没有其他类型)，“hello”是一个字符串(没有其他类型)，依此类推。 稍后[REF]我们将看到我们可以“细化”类型，这样一个值可以有多个细化类型:3可以是一个数、奇数，也可以是素数，依此类推。 在某些语言[REF]中，这些类型注释在程序运行之前被检查，因此您可以在运行程序之前了解潜在的错误。 在其他语言中，您只能在程序执行期间发现它们。 Pyret本身的目标是提供这两种模式，所以您可以选择对您的上下文最有意义的模式。

**3.5分步骤定义函数**

在编写函数时，分阶段编写是很有用的。 首先，给它起个名字，确保您理解它的类型，并编写一些小文档来提醒您的用户和读者，他们可能不熟悉您的功能——在几周内，这可能就是您! -这意味着什么。 例如，这里有一个函数，给定工作时间，计算相应的工资:

**有趣的小时工资(小时::数字)->数字:**

**医生:“按小时计算工资总额，包括加班费，每小时10美元。”**

**3.1 Example: Moon Weight**

Suppose we’re responsible for outfitting a team of astronauts for lunar exploration. We have to determine how much each of them will weigh on the Moon’s surface. We know how to do this—we saw the expression earlier [REF]—but it’s boring to write it over and over again. Besides, if we copy or re-type an expression multiple times, sooner or later we’re bound to make a transcription error.This is an instance of the DRY principle. Separately, correcting errors is itself an interesting computer science topic, which we address much later [REF].

When looking at our Moon weight calculations—say

100 \* 1/6

150 \* 1/6

90 \* 1/6

—we see that there are parts that are “fixed” and parts that are “changing”. The fixed parts are the ones we don’t want to have to repeat; the changing parts are the ones we have no choice about (and want the freedom to vary). It would be nice to make a package that makes this difference clear.

The way we’ll do it is to write a *function*. A function takes one or more *parameters*, which are the parts that vary. Specifically, the way we create a function is to

•  Write down some examples of the desired calculation.

•  Identify which parts are fixed (above, \* 1/6) and which are changing (above, 100, 150, 90...).

•  For each changing part, give it a name (say earth-weight), which will be the parameter that stands for it.

•  Rewrite the examples to be in terms of this parameter:

earth-weight \* 1/6

***Do Now!***

Why is there only one expression, when before we had many?

We have only one expression because the whole point was to get rid of all the changing parts and replace them with parameters.

•  Name the function something suggestive: e.g., moon-weight.

•  Write the syntax for functions around the expression:

•  **fun** <function name>(<parameters>):

•   <the expression goes here>

**end**

where the expression is called the *body* of the function.

Wow, that looks like a lot of work! But the end-product is really quite simple:

**fun** moon-weight(earth-weight):

earth-weight \* 1/6

**end**

We will go through the same steps over and over, and eventually they’ll become so intuitive that we won’t even remember that we actually took step*s* to get from examples to the function: it’ll become a single, natural *step*.

How do we use this? From Pyret’s point of view, moon-weight is just another operator just like num-expt or overlay. Thus:

moon-weight(100)

moon-weight(150)

moon-weight(90)

will produce the same answers as the expressions we began with, but we’re not going to make any mistakes in the formula due to copying or retyping.

**3.2 Example: Japanese Flag**

Let’s create another function. Remember our Japanese flag ([REF])? Each time we wanted a different-sized flag, we had to change the value of unit and re-run the whole program. Instead, we should create a function that generates Japanese flags.

How many parameters does this function need? Going back to our earlier code, we see that the only thing that really changes is unit. Everything else is calculated from that. Therefore, we should turn unit into a parameter, and keep the rest of the computation (which is already in terms of unit) intact:

**fun** japan-flag(unit):

bg-width = unit \* 3

bg-height = unit \* 2

circ-rad = 3/5 \* 1/2 \* bg-height

red-circ = circle(circ-rad, "solid", "red")

white-rect = rectangle(bg-width, bg-height, "solid", "white")

overlay(red-circ, white-rect)

**end**

This function body creates several local [REF] variables, and eventually produces the result of the overlay expression, which is the flag shape. We can therefore use it many times:

japan-flag(100)

japan-flag(200)

japan-flag(50)

without having to re-run the program between changes.Note that if the generated image is large, Pyret will replace the acutal image with a thumbnail version of it. Click on the thumbnail to see the full image.

**3.3 Tests: Keeping Track of Examples**

In each of the functions above, we’ve started with some examples of what we wanted to compute, generalized from there to a generic formula, turned this into a function, and then used the function in place of the original expressions.

Now that we’re done, what use are the initial examples? It seems tempting to toss them away. However, there’s an important rule about software that you should learn: *Software Evolves*. Over time, any program that has any use will change and grow, and as a result may end up producing different values than it did initially. Sometimes these are intended, but sometimes these are a result of mistakes (including such silly but inevitable mistakes like accidentally adding or deleting text while typing). Therefore, it’s always useful to keep those examples around for future reference, so you can immediately be alerted if the function deviates from the examples it was supposed to generalize.

Pyret makes this easy to do. Every function can be accompanied by a where clause that records the examples. For instance, our Moon weight can be modified to read:

**fun** moon-weight(earth-weight):

earth-weight \* 1/6

**where**:

moon-weight(100) **is** 100 \* 1/6

moon-weight(150) **is** 150 \* 1/6

moon-weight(90) **is** 90 \* 1/6

**end**

When written this way, Pyret will actually check the answers every time you run the program, and notify you if you have changed the function to be inconsistent with these examples.

***Do Now!***

Check this! Change the formula—for instance, replace the body of the function with

earth-weight \* 1/3

and see what happens.

Of course, it’s pretty unlikely you will make a mistake with a function this simple (except through a typo). After all, the examples are so similar to the function’s own body. Later, however, we will see that the examples can be much simpler than the body, as a result of which it’s no longer so easy to tell that they behave the same way, and we will find that it can be difficult to make the body match the examples. In fact, this is such a common in real software production that professional programmers always write down such examples—called *tests*—to make sure their programs are behaving as they expect.

**3.4 Type Annotations**

Suppose we were to call moon-weight on a string:

moon-weight("Armstrong")

***Do Now!***

What happens?

Pyret generates an error, saying that you can’t multiply a number by a string (whoever taught you arithmetic will be pleased to hear that).

In a function this small, it hardly matters. But if you had a much bigger function, it would be frustrating to get a similar error from deep in its bowels. Worse, if you get a function that someone else wrote, you need to read the entire function—which could be quite a bit larger—to figure out what kinds of values it consumes and produces.

Fortunately, we can do better. Pyret lets you write *annotations* on functions that indicate its values. Specifically, in the case of moon-weight, because it consumes and produces numbers, we would write:

**fun** moon-weight(earth-weight :: Number) -> Number:

earth-weight \* 1/6

**end**

We’ve left out the where examples for brevity, but you can write those too.Now, just by reading the function you can tell that it consumes a number (the :: Number part) and that it also produces one (the -> Number part).

***Do Now!***

What happens now when you run moon-weight("Armstrong")?

***Do Now!***

What would the annotations be on japan-flag?

Because japan-flag consumes a number and produces an image, we write:

**fun** japan-flag(unit :: Number) -> Image:

bg-width = unit \* 3

bg-height = unit \* 2

circ-rad = 3/5 \* 1/2 \* bg-height

red-circ = circle(circ-rad, "solid", "red")

white-rect = rectangle(bg-width, bg-height, "solid", "white")

overlay(red-circ, white-rect)

**end**

Observe that these annotations are clearly optional: until this section, our functions had neither. In fact, you can use annotations in one place and not another. Also, you can place annotations on any new variable, not only those in parameters: for instance, the variables inside japan-flag can also be annotated.

***Do Now!***

Fill in the annotations in each of the blanks:

**fun** japan-flag(unit :: Number) -> Image:

bg-width :: \_\_\_ = unit \* 3

bg-height :: \_\_\_ = unit \* 2

circ-rad :: \_\_\_ = 3/5 \* 1/2 \* bg-height

red-circ :: \_\_\_ = circle(circ-rad, "solid", "red")

white-rect :: \_\_\_ = rectangle(bg-width, bg-height, "solid", "white")

overlay(red-circ, white-rect)

**end**

The full-annotated function would be:

**fun** japan-flag(unit :: Number) -> Image:

bg-width :: Number = unit \* 3

bg-height :: Number = unit \* 2

circ-rad :: Number = 3/5 \* 1/2 \* bg-height

red-circ :: Image = circle(circ-rad, "solid", "red")

white-rect :: Image = rectangle(bg-width, bg-height, "solid", "white")

overlay(red-circ, white-rect)

**end**

***Do Now!***

Change one of the annotations to be incorrect: e.g.,

red-circ :: Number = circle(circ-rad, "solid", "red")

•  When do you get an error? Is it when you click Run or only when you actually use japan-flag?

•  Which part of your program does the error refer to?

The things we put in the annotations—Number, String, etc.—are called *types*. Types help us tell apart different kinds of data. Every value has a type, and no value has more than one type. Thus, 3 is a Number (and no other type), "hello" is a String (and no other type), and so on.Later [REF] we will see that we can “refine” types so that a value can have more than one refined type: 3 can be a number, an odd number, but also a prime number, and so on. In some languages [REF], these type annotations are checked *before* the program runs, so you can learn about potential errors before ever running your program. In other languages, you only discover them during program execution. Pyret itself aims to provide both modes, so you can choose whichever makes most sense for your context.

**3.5 Defining Functions in Steps**

When writing functions, it is useful to write it in stages. First, give it a name, make sure you understand its types, and write a little documentation to remind your user and reader, who may be unfamiliar with your function—in a few weeks, this could be *you*!—what it’s meant to do. For instance, here’s a function that, given a number of hours worked, computes the corresponding salary:

**fun** hours-to-wages(hours :: Number) -> Number:

**doc**: "Compute total wage from hours, with overtime, at $10/hr base"

文档中第 1 至 1000 字的翻译内容正在加载...

确定