让我们想想我们使用的一些程序。 我们可以使用搜索引擎谷歌:它使用搜索查询并生成搜索结果。 我们可能会使用Facebook，一个社交网络:它消耗我们的友谊信息，并从我们的朋友那里产生更新。 我们可以使用Amazon，一个商店:它消费我们感兴趣的东西的描述，并生成与我们的描述相匹配的产品列表。 我们有时使用weather .com，一个天气站点:它使用我们的位置并为该位置生成天气预报。 事实上，这些系统消耗并产生更多:它们消耗我们过去偏好的历史，并为相关产品生成广告、推荐(参考协同过滤)，等等。 简而言之，程序消耗并产生信息。

信息是一个模糊的概念; 计算机实际消耗和产生的是我们所说的数据。 这种差别很微妙，但很重要。 “我们的位置”是一个模糊的概念，但是一个气象站点实际消耗的是一个非常具体的表示，比如一个城市的名称(如“Providence, RI, USA”)或者它的GPS坐标(如“41.8236 N, 71.4222 W”)。 对于如何表示该信息，甚至可能有多种选择。 例如，有些系统可能代表普罗维登斯的GPS坐标，如下图所示:

|  |
| --- |
| 北纬41.8236度，西经71.4222度 |

其他人可能会选择用正数和负数来代表半球:

|  |
| --- |
| + 41.8236,-71.4222 |

您甚至可以想象通过一些巧妙的数字技巧(REF godel编码)将两者组合成一个数字，等等。 因此，这些是表示相同信息的不同类型的数据。

因此，为了编写程序，我们必须理解它们的数据。 我们将从理解最基本的数据类型开始，并用它们编写一些简单的程序。 随着我们的程序越来越复杂，我们将需要表示更多有趣的信息。 我们大部分的演讲都直接来自于《如何设计程序》这本书，你可以在http://htdp.org/上找到这本书。 (本段包含链接：[http://htdp.org/](http://htdp.org/" \t "_blank)  )

**2.1数字**

程序中最常见的一些数据是数值型的。 它们可能表示如上所示的GPS坐标，或者一个人的年龄，或者屏幕上图片的大小，或者搜索查询的结果数量。 数字在炮塔中很容易表示; 例如，这是布朗大学成立的年份:

1764

或者是一个让数论爱好者兴奋的数字: (本段包含链接：[number theory fans](https://en.wikipedia.org/wiki/1729_(number)" \t "_blank)  )

1729

不出所料，如果您向Pyret询问它们的值，Pyret会返回您输入的相同值:

|  |
| --- |
| > 1764 |
| 1764 |
| > 1729 |
| 1729 |

我们已经知道其值的程序有什么用? 稍后您会发现:(“测试价值”部分)。

当然，你可以用你能识别的方式组合数字; 我们称这些表达式为，就像在代数中一样:是的，你必须在运算前后加上空格，比如+。

|  |
| --- |
| > 1764 + 1729 |
| 3493 |
| > 1764 - 1729 |
| 35 |
| > 1764 \* 1729 |
| 3049956 |

如果您以前有过编程经验，您可能会有点担心乘大数的问题。 别担心，发疯吧:

|  |
| --- |
| > 1764 \* 1729 \* 1764 \* 1729 \* 1764 \* 1729 \* 1729 \* 1764 \* 1764 \* 1729 |
| 86531512776056800758948096 |

***现在做!***

示例中每个表达式只使用一个操作符。 如果我们试图合并多个不同的算子会发生什么?

在Pyret中，如果将操作符组合在一起，语言不想猜测您可能的意思(因为它可能猜错了):如果您编写

3 - 2 + 1

你是说3 -(2 + 1)还是(3 + 1)- 2? 它要求您编写括号(就像我们刚才所做的那样)来明确您的意图，而不是猜测。

当然，我们可以对数字执行许多操作，但是我们并没有为所有的数字都使用特殊的符号。 您可以使用Pyret中更多的数值运算符，使用您在数学课上学到的语法:数值平方根的运算符称为number -sqrt，您可以这样使用它:

|  |
| --- |
| > num-sqrt (4) |
| 2 |

如果操作符有多个参数，请用逗号分隔:

|  |
| --- |
| > num-expt (2、4) |
| 16 |

其中numexpt计算指数。

**锻炼**

您可能会有一个很自然的问题，“我是否需要在参数之间的逗号后面加上空格?” “问得好! 你自己去看看吧。

如果您有使用其他编程语言的经验，您现在可能会开始对不同类型的数字感到疑惑。 例如，在许多语言中，这个表达式

(1 / 3) \* 3

不幸的是，结果是零。 不要问。 或者，等会儿再问。 幸运的是，在Pyret中生成1。 这是因为Pyret以1 / 3的比例表示结果，正如您所期望的那样。 事实上，Pyret一直试图保持比率:如果你计算的话

(1764 \* 1729 \* 1764) / (1764 \* 1729 \* 1764 \* 1729)

Pyret将产生一个相当于1/1729的答案。 实际上，如果您单击Pyret生成的输出，您可以在表示相同数据的不同方式之间切换，其中一种是1/1729。

**锻炼**

如果除以0会发生什么?

1.  你还能指望什么?

2.  Pyret是做什么的?

**2.2表达式与值**

在我们继续之前，我们必须介绍一个术语。 值是表达式的结果:我们已经用它进行了计算，不能再做更多了。 因此，1764 \* 1729不是一个值，因为我们仍然可以执行乘法; 但是3049956是一个值，因为它没有更多等待执行的操作。

让我们来看另一个表达式的例子。 想象一个太空旅行者访问月球。 在月球上，这个宇航员的重量只有地球上的六分之一。 因此，如果它们的地球重量为50千克，它们的月球重量(以千克为单位)为

50 \* 1/6

如果他们的地球重量是150磅，他们的月球重量(以磅为单位)是

150 \* 1/6

一般来说，给定它们的地球重量，我们可以写一个表达式来计算它们的月球重量。 我们将很快回到这个例子[REF]。 注意，Pyret并不关心50代表公斤还是磅。 这些是单元，而单元不是编程语言的一部分。 这使得创建具有臭名昭著历史的单元错误变得很容易。 稍后[REF]我们将看到如何在程序中编写一些单元处理来减少此类错误。 (本段包含链接：[infamous history](http://spacemath.gsfc.nasa.gov/weekly/6Page53.pdf" \t "_blank)  )

***现在做!***

在上一节和这一节中我们写的分数中你看到什么有趣的东西了吗?

这很微妙，所以如果你没听清也不用担心:我们把1/1729写成了1/1729，把1/6写成了1/6。 难道不是所有的算术运算(比如/)都应该被空格包围吗?

这是因为1/1729不是一个表达式，它是一个数值——确切地说，是一个数字本身。 也就是说，1 / 1729是一个表达式; 仍然需要做一些工作(除法)来获得一个值。 相反，Pyret写数字的方法对于直接写有理数有一个特殊的限制:你可以写1/1729就像你可以写1和1729一样好，而且它的值也一样大。

**2.3变量的名称**

通常，引用某个东西的名称比它的值更方便。 例如，六个月后如果你在一个程序中看到1764，你可能不记得那是什么意思; 但如果你看到的是建立布朗基金会，你就会有很好的判断力。 Pyret让我们给值命名:

brown-founding = 1764

今年= 2016

然后我们可以在表达式中使用这些名称:

今年,brown-founding

实际上，我们不必只命名值; 我们甚至可以命名表达式:

布朗年龄=今年-布朗成立

我们把这些名字叫做变量，就像代数里一样。 我们将使用“绑定”这个词，这里的意思是“绑定”，因为变量现在绑定到那个值。 要引用名称与值的关联:也就是说，brown-开国一定要追溯到1764年。

**2.4字符串**

当然，计算机处理的不仅仅是数字。 我们可能想写下一个城市的名字，一个人的名字，文件的一部分，等等。 在Pyret中，字符串用来表示这样的文本:

“普罗维登斯,国际扶轮”

“印度班加罗尔”

"敏捷的棕色狐狸跳过了懒狗"

字符串以双引号开始和结束; 注意，您必须在键盘上使用双引号(")，而不是由文字处理程序生成的开引号和闭引号("和")。 你也可以用单引号(')代替:

普罗维登斯,国际扶轮

“印度班加罗尔”

敏捷的棕色狐狸跳过了懒狗

当然，如果字符串本身包含双引号，您可能想知道Pyret如何判断字符串的结束位置:

|  |
| --- |
| 这本书名叫《结构与解释》 |

***现在做!***

当你输入这个时会发生什么?

这是两个字符串(“The book was called”和“”)，其中包含一些变量(比如结构)和其他中间内容吗? 不用说，派瑞特很困惑。 那么如何输入呢? 有两种简单的方法。 首先，我们可以使用其他引号来避免混淆:

这本书叫做《结构与诠释》

或者(当Pyret打印上面的值时，您可能已经注意到了)，您可以告诉Pyret“this quote is part of the string, not the end of it”:

这本书名叫《结构与解释》。

两种方法都可以; 选择可读性更强、更不容易引起混淆的。

**2.4.1多行字符串**

有时使用跨多行的字符串很方便。 通常，超过一行的字符串表示程序中的错误(因为程序员忘记关闭字符串)。 因此，如果您试图在您的程序中编写一个多行字符串，Pyret将产生一个错误:

|  |
| --- |
| “那我们走吧，你和我， |
| 当夜幕降临天空" |

***现在做!***

试一下，看看结果是什么错误。 熟悉它!

相反，Pyret对多行字符串使用非常不同的语法:

“那我们走吧，你和我，

当夜晚在天空中展开

使用三回勾是您向Pyret发送信号的方式，“我真的希望它跨越多行”。 注意，当Pyret打印这个字符串时，它用\n替换换行。 实际上，你可以把这个多行字符串都写在一行上

“那么，我们走吧，你和我，当夜幕降临在天空的时候。”

同样，这两个是等价的，您可以选择更方便的。

**2.4.2字符串操作**

当然，我们不仅可以创建字符串，还可以用它们执行计算——本质上是“用字符串进行算术”。 例如，我们可以把它们连接起来:

Let’s think about some of the programs we use. We might use Google, a search engine: it *consumes* a search query and *produces* a search results. We may use Facebook, a social network: it *consumes* information about our friendships and *produces* updates from our friends. We could use Amazon, a store: it *consumes* descriptions of things we’re interested in and *produces* lists of products that match our descriptions. We sometimes use Weather.com, a weather site: it *consumes* our location and *produces* a weather forecast for that location. In fact these systems consume and produce even more: they consume our history of past preferences, and produce ads, recommendations for related products [REF collaborative filtering], and so on. In short, programs consume and produce *information*.

Information is a fuzzy term; what computers actually consume and produce are what we call *data*. The difference is subtle but important. “Our location” is a vague concept, but what a weather site actually consumes is a very concrete *representation* of it, such as the name of a city (like “Providence, RI, USA”) or its GPS coordinate (like “41.8236 N, 71.4222 W”). There might even be multiple choices for how to represent that information. For instance, some systems might represent Providence’s GPS coordinates as we have shown above:

|  |
| --- |
| 41.8236 N, 71.4222 W |

Others might choose to use positive and negative numbers to represent the hemisphere:

|  |
| --- |
| +41.8236, -71.4222 |

You could even imagine combining the two into a single number through some clever numeric trick [REF gödel encoding], and so on. Therefore, these are different kinds of data to represent the same information.

Therefore, to write programs, we have to understand their data. We will begin by understand the most basic kinds of data, and writing some simple programs with them. As our programs get more sophisticated, we will need to represent more interesting kinds of information.Much of our presentation is directly derived from the book *How to Design Programs*, which you can find at http://htdp.org/.

**2.1 Numbers**

Some of the most common data in programs are numeric. They might represent a GPS coordinate as above, or a person’s age, or the size of a picture on a screen, or the number of results to a search query. Numbers are easy to represent in Pyret; for instance, here’s the year Brown University was founded:

1764

Or a number that excites number theory fans:

1729

Unsurprisingly, if you ask Pyret for their values, Pyret gives you back the same value you entered:

|  |
| --- |
| > 1764 |
| 1764 |
| > 1729 |
| 1729 |

What use are programs whose values we already know? You’ll find out later: (part "values-for-tests").

Of course, you can combine numbers in ways you will recognize; we call these *expressions*, just as in algebra:Yes, you must put spaces around operations like +.

|  |
| --- |
| > 1764 + 1729 |
| 3493 |
| > 1764 - 1729 |
| 35 |
| > 1764 \* 1729 |
| 3049956 |

If you have prior programming experience, you may worry a little about multiplying big numbers. Don’t worry, go wild:

|  |
| --- |
| > 1764 \* 1729 \* 1764 \* 1729 \* 1764 \* 1729 \* 1764 \* 1729 |
| 86531512776056800758948096 |

***Do Now!***

The examples use only one operator per expression. What happens if we try to combine multiple different operators?

In Pyret, if you combine operators, the language doesn’t want to guess what you might have meant (because it might guess wrong): if you write

3 - 2 + 1

did you mean 3 - (2 + 1) or (3 + 1) - 2? Rather than guess, it asks that you write parentheses (as we just did) to make your intent clear.

Of course, there are *many* operations we can perform on numbers, and we don’t have special symbols for all of them. You can use Pyret’s many more numeric operators using the syntax you recognize from math classes: the operator for numeric square root is called num-sqrt and you use it as follows:

|  |
| --- |
| > num-sqrt(4) |
| 2 |

If you have multiple parameters to the operator, separate them with commas:

|  |
| --- |
| > num-expt(2, 4) |
| 16 |

where num-expt computes the exponential.

**Exercise**

A natural question you might have is, “Do I need to put spaces after the commas between parameters?” Good question! Go ahead and check for yourself.

If you have experience with other programming languages, you might now start to wonder about different kinds of numbers. For instance, in many languages, this expression

(1 / 3) \* 3

unfortunately results in zero.Don’t ask. Or, well, ask later. Fortunately, in Pyret this produces 1. That’s because Pyret represents the result of 1 / 3 exactly as a ratio, just as you’d expect. In fact, Pyret tries to preserve ratios at all times: if you compute

(1764 \* 1729 \* 1764) / (1764 \* 1729 \* 1764 \* 1729)

Pyret will produce an answer that is equivalent to 1/1729.In fact, if you click on the output produced by Pyret, you can switch between different ways of *presenting* the same datum, one of which is 1/1729.

**Exercise**

What happens if you try to divide by zero?

1.  What do you expect?

2.  What does Pyret do?

**2.2 Expressions Versus Values**

Before we proceed, we have to introduce one piece of terminology. A *value* is the result of an expression: we are done computing with it and can’t do any more. Thus, 1764 \* 1729 is *not* a value, because we can still perform the multiplication; but 3049956 is a value because it has no more operations waiting to be done.

Let’s look at another example of expressions. Consider a space traveler visiting the Moon. On the Moon, this astronaut weighs only one-sixth their weight on earth. Thus, if their Earth weight is 50kg, their Moon weight (in kilograms) is

50 \* 1/6

If their Earth weight is 150lb, their Moon weight (in pounds) is

150 \* 1/6

In general, given their Earth weight we can write an expression to calculate their Moon weight. We’ll return to this example soon [REF].Note that Pyret does not care whether 50 represents kilograms or pounds. These are *units*, and units are not part of the programming language. This makes it easy to create unit errors, which have an infamous history. Later [REF] we will see how we can write some unit processing into our programs to reduce such errors.

***Do Now!***

Do you see something curious in the fractions we’ve written in the previous section and this one?

This is subtle, so don’t worry if you didn’t catch it: we wrote 1/1729 instead of 1 / 1729, and 1/6 instead of 1 / 6. Weren’t all arithmetic operations (like /) supposed to be surrounded by spaces?

That’s because 1/1729 is not an expression, it’s a value—specifically, a number—in itself. That is, 1 / 1729 is an expression; there is still work that needs to be done (the division) to obtain a value. In contrast, Pyret’s way of writing numbers has a special allowance for writing rational numbers directly: you can write 1/1729 just as well as you can write 1 and 1729, and it’s just as much of a value.

**2.3 Variables to Name Values**

It’s often more convenient to refer to the name of something than its value. For instance, six months from now if you saw 1764 in a program, you might have no recollection of what that means; but if you instead saw brown-founding you’d have a pretty good sense. Pyret lets us give names to values:

brown-founding = 1764

this-year = 2016

We can then use the names in expressions:

this-year - brown-founding

Indeed, we don’t have to name only values; we can even name expressions:

brown-age = this-year - brown-founding

We call these names *variables*, just as in algebra. We’ll use the word *bound*“Bound” here means in the sense of “tied down”, because the variable is now tied down to that value. to refer to the association of names with values: that is, brown-founding is bound to 1764.

**2.4 Strings**

Of course, computers process much more than just numbers. We might want to write down the name of a city, the name of a person, parts of a document, and so on. In Pyret, *strings* are used to represent such text:

"Providence, RI"

"Bangalore, India"

"the quick brown fox jumped over the lazy dog"

Strings begin and end in double-quotes; note that you have to use the double-quotes on your keyboard (**"**), not opening- and closing-quotes (“ and ”) as generated by word processors. You can also use single-quotes (**'**) instead:

'Providence, RI'

'Bangalore, India'

'the quick brown fox jumped over the lazy dog'

Naturally, you might wonder how Pyret can tell where a string ends if the string itself contains a double-quote:

|  |
| --- |
| "The book was called "Structure and Interpretation"" |

***Do Now!***

What happens when you enter this?

Is that two strings ("The book was called " and "") with some variables (like Structure) and other stuff in-between? Needless to say, Pyret gets confused. So how *do* you enter this? There are two easy ways. First, we can use the other quotes to avoid confusion:

'The book was called "Structure and Interpretation"'

Or (as you may have noticed when Pyret prints the above value), you can tell Pyret “this quote is part of the string, not the end of it”:

"The book was called \"Structure and Interpretation\""

Either will work fine; pick whichever is more readable and less likely to cause confusion.

**2.4.1 Multi-Line Strings**

It’s sometimes convenient to have a string that spans multiple lines. Usually, strings that go beyond one line represent an error in the program (because the programmer forgot to close the string). Therefore, if you try to write a multi-line string in your program, Pyret will produce an error:

|  |
| --- |
| "Let us go then, you and I, |
| When the evening is spread out against the sky" |

***Do Now!***

Try this out and see the resulting error. Become familiar with it!

Pyret instead uses a very different syntax for multi-line strings:

```Let us go then, you and I,

When the evening is spread out against the sky```

Using the triple-back-tick is your way of signaling to Pyret, “I really do want this to span multiple lines”. Notice that when Pyret prints this string, it replaces the newline with \n. Indeed, you could have written this multi-line string all on line line as:

"Let us go then, you and I,\nWhen the evening is spread out against the sky"

Again, the two are equivalent, and you can choose whichever is more convenient.

**2.4.2 Operations on Strings**

Of course, we can not only create strings, we can also perform computations with them—essentially, “arithmetic with strings”. For instance, we can *concatenate* them:

文档中第 1 至 1000 字的翻译内容正在加载...

确定