

WorkScript



* 语法简单，学习成本低
* 设计严谨，不失专业性
* 精心优化，运行速度快
* 可能是全世界最简单的编程语言

荆佳颉 主编

一入门就精通

内部读物

前言

从50年代至今，计算机已经越来越贴近人们的生活。随着数以亿计的APP，网站和企业软件日复一日地涌现，我们意识到计算机在人类社会的发展中越来越占有举足轻重的地位。大部分人使用Excel进行数据的处理，使用在线工具来解决各自的需求。但是，这些应用程序的功能都是单一固定的。当我们的想法发生改变时，应用软件往往不能满足我们丰富多彩的想法。越来越多的人出于各种各样的想法，希望自己能够掌握一门编程语言，与计算机进行自由的沟通。然而尽管如此，大部分人依然认为编程是一件困难的事情。

人类从未停止追求编程的简单化，全民化。而WorkScript正是这个历史进程中的一个里程碑式的作品。它不是为了计算机的专业人员而设计的，相反，它是为了所有希望走近编程的人而诞生的。所以，WorkScript生来具备简单的语法和易懂的逻辑。同时，作为一款编程语言，它的内部隐藏了非常多的精妙思想和专业设计，在保证功能完整，专业，强大的同时，为大众提供了简单的解决方案。我们希望这个作品可以得到大家的喜爱，并能够对编程的大众化趋势起到有力的推进作用。

荆佳颉 写于 北京工业大学

2018年9月7日

目录

[第一章：快速上手 1](#_Toc1689809309)

[1.1 获取WorkScript 1](#_Toc1690883452)

[1.2 安装与启动WorkScript 1](#_Toc1272945631)

[1.3 编写第一个程序 3](#_Toc1484459269)

[1.4 少废话，先看东西 6](#_Toc1544543368)

[1.4.1 计算两个数字的和 6](#_Toc1315259463)

[1.4.2 取数字的绝对值 6](#_Toc1119671142)

[1.4.3 判断两个值是否相等 7](#_Toc604752474)

[1.4.4 计算斐波那契数列某一项 7](#_Toc493073395)

[1.4.5 计算任意个数字的最大值 8](#_Toc1709022339)

[1.5 注释 8](#_Toc1305801021)

[第二章：变量和值 10](#_Toc798439299)

[2.1 值 10](#_Toc1020220627)

[2.1.1 数字 10](#_Toc1564947176)

[2.1.2 字符串 10](#_Toc1902143153)

[2.1.3 布尔值 11](#_Toc1218360320)

[2.2 变量 12](#_Toc2094463588)

[2.2.1 变量的一般特性 12](#_Toc329124428)

[2.2.2 变量的类型约束 12](#_Toc702133519)

[第三章：一般运算 14](#_Toc1344690178)

[3.1 四则运算 14](#_Toc1634546306)

[3.2 赋值运算 15](#_Toc1778257415)

[3.3 比较运算 16](#_Toc394691752)

[3.4 赋值和判等的歧义 17](#_Toc1670448568)

[第四章：函数 20](#_Toc798829140)

[4.1 函数的介绍 20](#_Toc2124459166)

[4.2 函数的组成 22](#_Toc796268084)

[4.2.1 函数名 22](#_Toc1772802742)

[4.2.2 函数的参数 22](#_Toc2015367351)

[4.2.3 函数的约束 24](#_Toc1376381551)

[4.2.4 函数的实现 27](#_Toc839018555)

[4.2.5 函数的返回值 28](#_Toc1557693012)

[4.3 函数的调用 29](#_Toc919781355)

[4.3.1 按参数顺序传递参数的函数调用 29](#_Toc2111964186)

[4.3.2 按参数名传递参数的函数调用[尚未实现] 29](#_Toc894668633)

[4.3.3 匿名函数的调用 30](#_Toc316841075)

[4.4 函数的性质 31](#_Toc1279740002)

[4.4.1 函数的重载 31](#_Toc2014339775)

[4.4.2泛化函数与特化函数 32](#_Toc921593550)

[4.4.3 递归函数 32](#_Toc1772813397)

[4.4.4 匿名函数 33](#_Toc1575878467)

[4.4.5 偏函数 [尚未实现] 35](#_Toc79910923)

[4.4.6 回调函数 36](#_Toc423769048)

[4.4.7 闭包函数 [尚未实现] 37](#_Toc448615446)

[4.4.8 纯函数 38](#_Toc1644858100)

[4.4.9 外部函数 40](#_Toc178428553)

# 第一章：快速上手

## 1.1 获取WorkScript

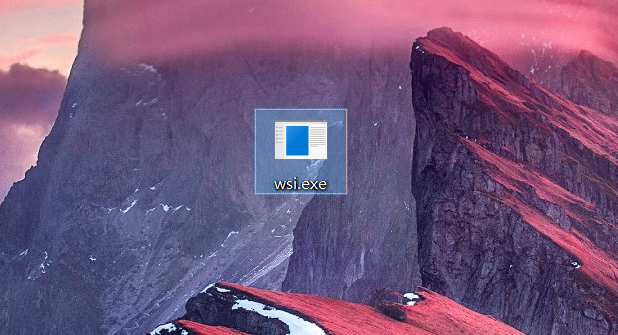
WorkScript官方网站：<http://www.workscript.org>

您可以在这里下载WorkScript解释器wsi (WorkScript Intepreter) 以及WorkScript的文档教程。

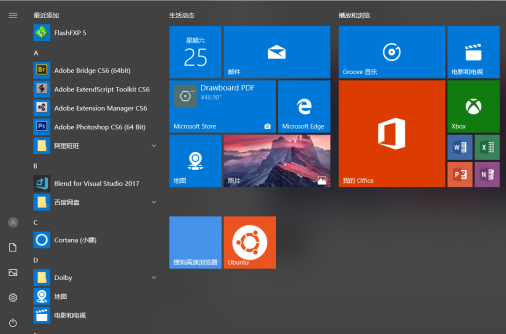
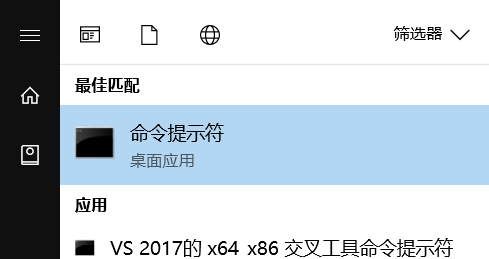
## 1.2 安装与启动WorkScript

目前的WorkScript还图形化编辑界面，仅有命令行工具wsi，所以，启动WorkScript是一件略显麻烦的事情，但同时也是一项非常有意义的工作。

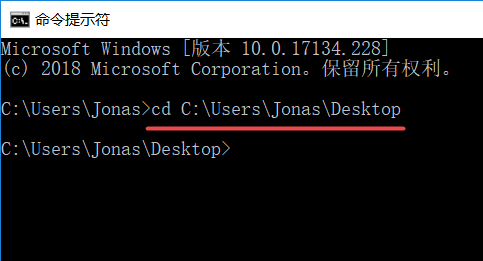
首先获取到WorkScript解释器的程序wsi.exe，放置在任意你想要的位置。作为示例，我将其放置在桌面上。



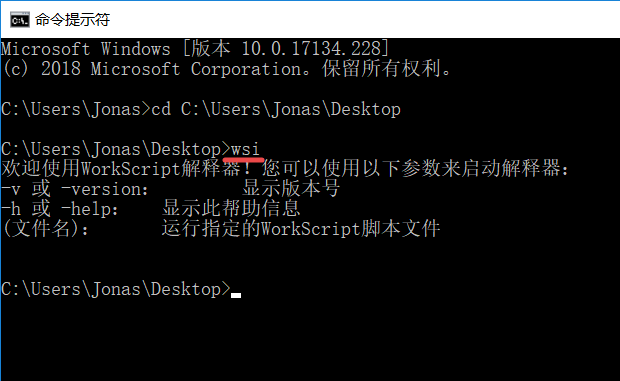
接下来，进入命令行。我们点击Windows桌面左下角的开始图标，打开开始菜单界面，直接输入”cmd”即可找到命令行。

进入命令行之后，我们将当前目录切换到wsi所在的目录。以上图为例，我们使用cd命令进行路径的切换到桌面目录。



目录切换完成之后，我们在当前目录直接输入wsi（或者wsi.exe，在Windows下允许省略exe后缀），即可启动WorkScript解释器。

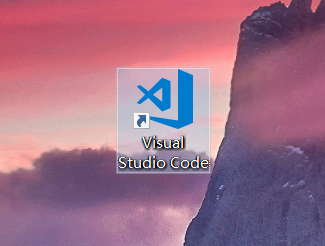


可以看到，不同于其他晦涩难懂的编程语言，WorkScript解释器的提示语是以中文显示的。这是因为WorkScript面向的对象是非计算机专业的人士。WorkScript语言的设计一直是以简单，易懂为基本的设计原则，无论是在语法的设计上，还是在逻辑思维上，WorkScript都尽量贴近没有学过编程的人，在不失专业性的同时，为更加广大的群体提供了与电脑沟通的语言。

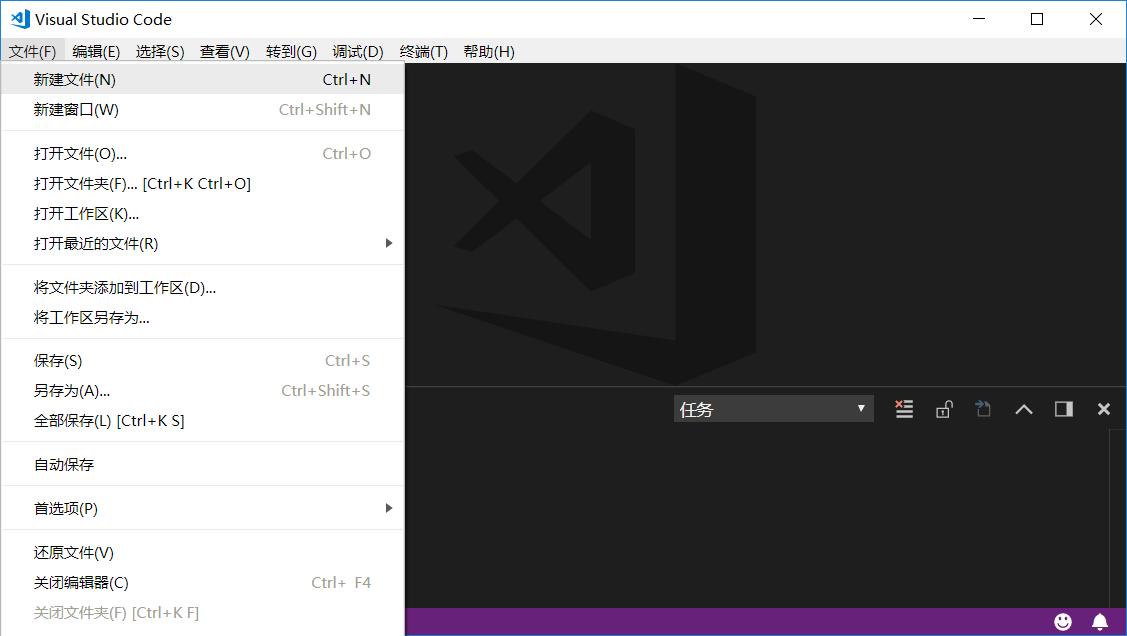
## 1.3 编写第一个程序

让我们使用WorkScript来打印一行文字”It works!”。我们可以选择一款普通的文本编辑器来编辑WorkScript代码，例如Windows的“记事本”。不过这里我们为大家推荐微软公司的”VS Code”编辑器。

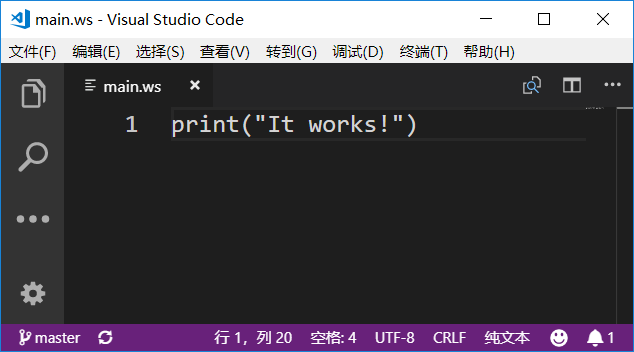
关于VS Code的安装这里不再介绍，读者可以从百度自行下载安装最新版本的VS Code。这里我们以安装好的VS Code为例：



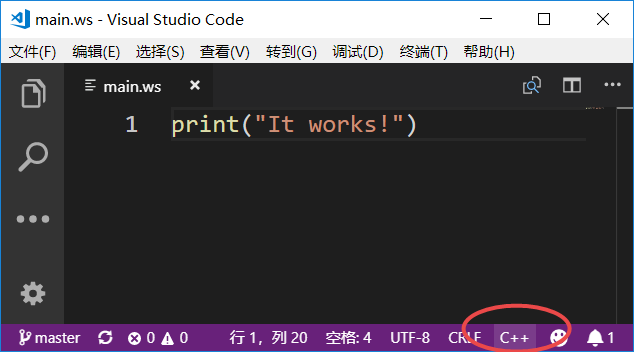
打开VS Code，新建一个文件：



在新建的文件中输入如下WorkScript代码：



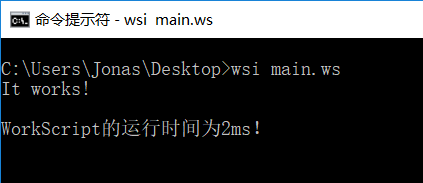
我们发现，我们输入的文字并没有颜色。我们可以点击右下角的“纯文本”，将其更换为“C++”,这样VS Code可以为WorkScript代码上色。



接下来将文件保存为”.ws”后缀的文件。这里我们保存为”main.ws”放置在桌面上。如图：



接下来让我们用之前介绍的WorkScript解释器来运行它。如何启动WorkScript解释器这里不再重复。我们使用解释器来运行main.ws的结果如下图所示：

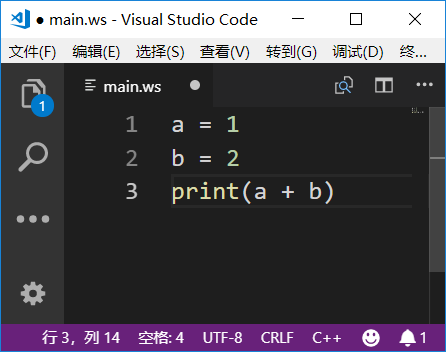
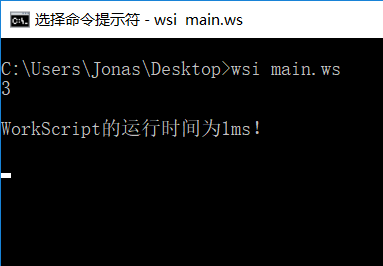


如果操作无误，读者应该可以看到”It works!”的输出。这就意味着WorkScript程序执行成功了。

## 1.4 少废话，先看东西

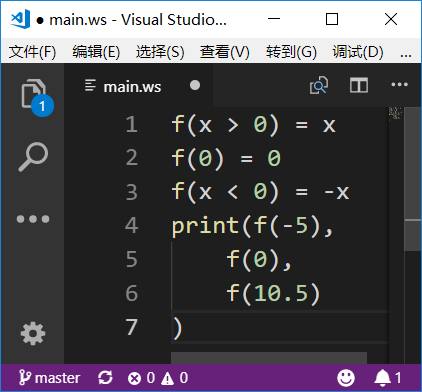
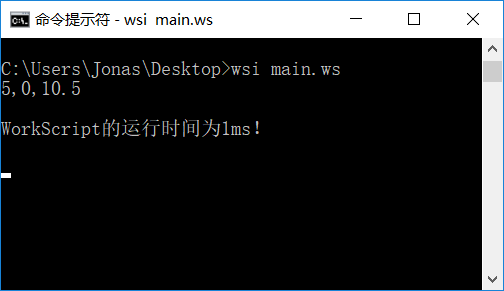
说的好不如做的好，在下一章正式开始介绍WorkScript之前，我们先看一些简单的WorkScript程序，对WorkScript形成一个大体的认识。

### 1.4.1 计算两个数字的和

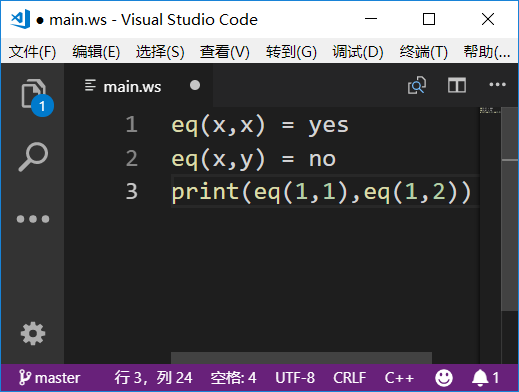
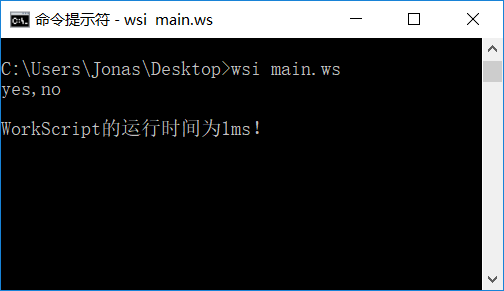
上例的1，2行主要展示了变量的声明和值的运算，我们将在第2，3章进行详细的讨论。第三行”print()”是函数的调用，我们将在第4章进行相关的介绍。

### 1.4.2 取数字的绝对值

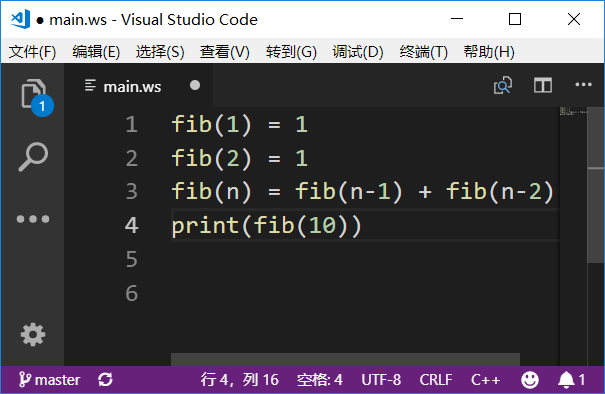
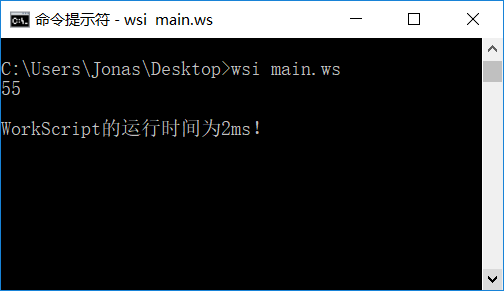
上例中的前3行是声明了取绝对值的函数f，并且对于参数x的三种不同的情况，进行了不同的声明。关于函数的声明，我们会在第4章进行介绍。第4行使用print函数分别打印出-5, 0, 10.5三个数字的绝对值，不出意外，我们看到了5, 0, 10.5三个数字。

### 1.4.3 判断两个值是否相等

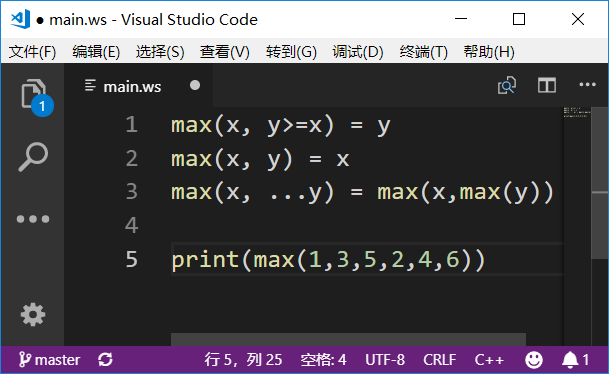
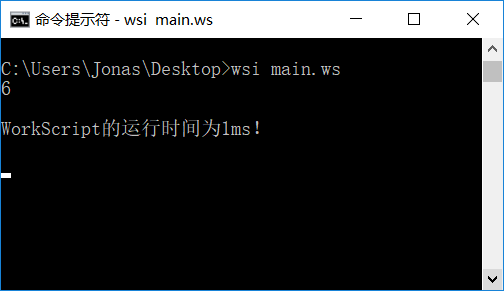
前两行声明的eq函数接收两个参数，其中第一行的两个参数x, x相同，而第二行的两个参数x, y不要求相同。所以eq(1,1)对第一行成立，得到yes。eq(1,2)对第一行不成立，对第二行成立，得到no。其中yes和no是两种布尔类型的值，我们会在第2章进行详细介绍。

### 1.4.4 计算斐波那契数列某一项

这是一个递归函数的示例，fib(n)在计算的过程中用到了fib(n-1)和fib(n-2)，从而形成了递归。要了解递归，你需要先了解递归，我们会在第5章对递归函数进行详细的介绍。

### 1.4.5 计算任意个数字的最大值

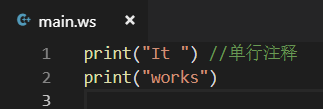
 

这是一个变参函数的示例。前两行的函数声明和之前的示例类似，不再重复讲解。第3行的…y是一个变参，带有…表示的参数可以接收多个值。例如上面的x, …y接收到参数1,3,5,2,4,6后，x的值为1，y的值为3,5,2,4,6。后面的执行过程和上面介绍的递归函数类似，不再详细解释。

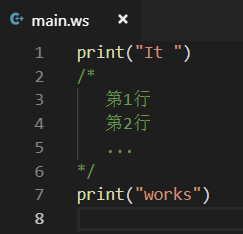
## 1.5 注释

在程序中我们常常希望记录一些备注性的文本，方便程序的阅读。WorkScript支持两种形式的注释：单行注释和多行注释。

单行注释形式如下：

单行注释以//开头，随本行结束而结束。通常适合写一些简单短小的注释内容。

多行注释如下:

多行注释以/\*开始，以\*/结束，可以跨越多行，也可以放在一行的中间，或者放置在程序的任意位置。多行注释往往用来记录一些较长的内容，例如函数的详细介绍，作者和编写时间等内容。

注释不会影响程序的运行，对于程序来说，注释是不存在的。它们会在程序运行之前被解释器忽略掉。一般来说，我们要保持书写注释的习惯，让我们的程序可读性更好，同时也便于他人修改和维护我们的程序，尤其是在多人协作的软件编写时，注释就变得更加重要。

# 第二章：变量和值

## 2.1 值

在上一章，我们已经和值有过初步的接触了。和所有其他计算机语言一样，目前版本的WorkScript提供了以下几种类型的值：

***数字，字符串，布尔值***

函数作为一种特性非常丰富的值，我们在下一章单独进行讲解。下面我们依次为大家讲解其他几种值及相关的运算。

### 2.1.1 数字

数字是我们生活中最常见的值了。相信只要学过数学的人，都会知道数字有整数，小数，正数，负数，二进制数，十六进制数等等诸多分类。在WorkScript中，所有的数字用常见的十进制表示。如下都是合法的数字：

|  |
| --- |
| 1 100.5 0 -25 -30.6 +1 +14.8 |

### 2.1.2 字符串

字符串是计算机处理文字所使用的。所有的计算机语言都有字符串这个类型的值，但如果你是初次接触计算机语言，也许对字符串会感到陌生。那么下面的这个例子，就是一个标准的字符串：

|  |
| --- |
| “It works!” |

可以看到，字符串其实就是一段文字。在WorkScript中，字符串需要用双引号将文本括起来。

### 2.1.3 布尔值

说到布尔值，如果你是第一次接触计算机语言，可能会感觉到陌生。但其实它就存在于我们的生活中。例如，你的朋友问你是坐地铁上班的吗？你的回答可能是**“是”**或者**“否”**。老师问你“1+1”等于2对吗？你的回答可以是**“对”**或者**“错”**。在计算机的世界，我们将是，对，好等积极的回答统称为**“真值”。**将否，错，不好等反面的回答统称为**“假值”。**

在WorkScript中，我们定义了如下四种真值：

|  |
| --- |
| true yes ok good |

和如下三种假值：

|  |
| --- |
| false no bad |

其中，所有的真值都是相等的，所有的假值也都是相等的。也就是说，在计算中，yes=true=ok=good，false=no=bad。所以，使用哪种值不影响程序的运行结果，但应根据语境选择更加合适的词语。

## 2.2 变量

### 2.2.1 变量的一般特性

变量是个熟悉的概念了，无论是在数学上，还是在计算机语言中，都存在变量的概念。变量拥有名字，称为变量名。每个变量可以存储一个值，例如数字，布尔值，字符串以及函数等。并且在需要的时候可以从变量中获得存储的值。WorkScript中变量的名字必须符合如下规定：

***必须由字母，数字，下划线，中文或其他语言的文字组成。并且第一个字符不能是数字。***

也就是说，如下的变量名是合法的：

|  |
| --- |
| a \_b a1 b2\_ 员工 成绩1 |

而下列变量名是非法的：

|  |
| --- |
| 01 1a %b ~员工 <成绩> |

变量是用来存储值的，我们可以对变量进行赋值：

|  |
| --- |
| X = 1  Y = “hello world”  Z = true |

需要注意的是，WorkScript属于“静态类型语言”，变量一旦被赋值，其所拥有的类型也随之确定。例如，当X=1时，变量X的类型就是数字类型，此时如果再对X赋予其他类型的值，如X=”hello world”，是不合法的。

### 2.2.2 变量的类型

对于一个普通的变量x，我们可以为它指定类型，例如：

|  |
| --- |
| int x = 1 |

此时变量x的类型就确定为int类型。也许上面的例子比较简单，但是当程序复杂后，这样的错误会一个接一个顺着程序传递下去，直到很远的地方才会被发现。而发现错误后想要追根溯源找到这里，会浪费大量的时间。下面我们用类型约束为x绑定数字类型的值：

|  |
| --- |
| number x = 1  y = x + 1 |

这个时候，如果x存储的值不是数字类型，那么会直接引起一个运行时错误，从而令我们可以直接发现这个错误。

不论变量处于任何地方，都可以进行类型约束。例如上面例子中的y也可以添加类型约束:

|  |
| --- |
| number y = x + 1 |

# 第三章：一般运算

## 3.1 四则运算

四则运算和数学中的四则运算基本上没有区别，所以这里很简单的进行介绍。WorkScript提供了加法，减法，乘法，除法，取模（取余数），乘方等运算。我们可以用如下的运算符分别来实现上述计算：

|  |
| --- |
| a + b 加法  a – b 减法  a \* b 乘法  a / b 除法  a % b 取模 |

上面例子中的a,b和x都是变量名。如果直接使用值进行计算同样是可以的，例如1 + 2。

## 3.2 赋值运算

对于变量，我们往往需要进行赋值。在第二章中我们已经涉及到了对变量的赋值，在这里我们将对赋值操作进行具体的介绍。我们可以对变量进行赋值，也可以对计算结果为变量的运算进行赋值（例如函数返回了变量，我们在后面的章节中会进行讨论）。

赋值的语法如下：

|  |
| --- |
| 变量 = 表达式 |

或者使用Pascal语言风格的赋值符号：

|  |
| --- |
| 变量 := 表达式 |

其中表达式可以是变量，也可以是值，或者四则运算等。但是要注意的是，上面被赋值的必须是变量，不能是值，或者计算结果不是变量的表达式。例如下面的赋值是非法的：

|  |
| --- |
| 2 = 1 |

显然，2不是一个变量，它不能被赋值。

## 3.3 比较运算

当我们需要比较两个值的大小时，我们可以使用比较运算。WorkScript提供的比较运算有：大于，大于等于，小于，小于等于，等于，不等于。它们的写法如下：

|  |
| --- |
| a > b a大于b  a >= b a大于等于b  a < b a小于b  a <= b a小于等于b  a = b a等于b  a == b (C语言风格的判等符号，与上一行含义相同)  a != b a不等于b（或 a <> b） |

比较运算的结果是布尔值true或者false。例如如果执行print(2>1)会打印true，显然2大于1。如果执行print(2<1)会打印false，因为2并不小于1。如果执行print(2=1)会打印false，因为2也不等于1。

也许细心的读者已经发现了，在“赋值运算”中我们刚刚举过一个反例“2=1”，2不是一个变量，不能被赋值。为什么在这里却出现了“2=1”这样的表达式呢？原因是，上一小节的等号是赋值的含义，而在这一小节中，等号是判等的含义。虽然2不是变量，不能被赋值，但是作为一个值，它是可以用来判等的。那么如何区分等号到底是用作赋值还是判等呢？我们在下一小结对此进行讨论。

## 3.4 赋值和判等的歧义

在上一小节提到，由于赋值和判等两种运算共用等号“=”，导致等号出现了歧义的问题。这看起来是一个棘手的问题，下面让我们看看，在历史的长河中，语言设计者们曾经绞尽脑汁地想到过哪些糟糕的主意：

**C语言：**

|  |
| --- |
| a = b 赋值  a == b 判等 |

**Pascal：**

|  |
| --- |
| a := b 赋值  a = b 判等 |

**Prolog：**

|  |
| --- |
| A is B 赋值  A = B 判等 |

对于众多编程语言，在这里就不一一列举了。总之，没有一款编程语言让用户能够以一种正常的方式去完成这两件事，直到一向创造优秀产品的微软为我们设计的Visual Basic语言的问世：

**Visual Basic:**

|  |
| --- |
| a = b 赋值  a = b 判等 |

我们惊奇的发现在Visual Basic中，人们终于能够以习惯的方式来进行赋值和判等了。但是，这两者的歧义是如何被解决的呢？由于WorkScript的语法和Visual Basic这门传统语言的语法有着天差地别，故感兴趣的读者可以自行翻阅Visual Basic的语法。在这里我们介绍WorkScript语言的规则：

***当表达式处于独立位置的时候，它被认为是赋值表达式。***

***当表达式处于非独立位置的时候，它被认为是判等表达式。***

什么是非独立位置呢？一般的，如果一个表达式被另一个表达式所嵌套，那么它就是非独立位置，例如常见的判等表达式：

***print(a=b)*** 等号所在的表达式被*print()*表达式所嵌套，它是判等表达式。

***f(x) = 0 when x = 0*** 这也是一个函数声明，但是*x=0*位于函数的约束中。这也是一个非独立位置，视为判等表达式。

***f(x) = (x = 1)*** 这也是一个函数声明，此处x=1嵌套于括号中，故视为判等表达式。

什么是独立位置呢？一般的，如果表达式不被其他表达式所嵌套，或者直接嵌套于函数中，则它是独立表达式。例如常见的赋值表达式：

***a = b*** 这显然是一个赋值表达式。

**f(x) = {**

**a = x**

**}**

在这里，a=x被嵌套在f(x)中，是赋值表达式。

**f(x) = a = x** 这种情况有些极端，它和上面一行是一模一样的，仅仅省略了大括号，故是赋值表达式。

上面的规则已经详细的说明了赋值和判等表达式的判定。虽然看起来有些复杂，但在实际的使用中是非常自然的，以至于你甚至感受不到赋值和判等所带来的歧义。最后，值得一提的是，如果你真的认为在有些地方你无法确定到底是赋值还是判等，没有关系，我们可以使用下面的WorkScript标准赋值表达式和标准判等表达式来避免不必要的麻烦：

|  |
| --- |
| a := b 赋值  a == b 判等 |

# 第四章：函数

## 4.1 函数的介绍

由于函数是个众人皆知的概念，所以我们在前几章已经无数次地提到过函数。但是在这一章，我们将对函数进行详细的讨论。

在正式介绍函数之前，我们先用一个简单的例子来感受一下函数。我们使用WorkScript设计一套评分系统，我们假设一个人的成绩区间是0到100分，其中0-60为不合格 ，60-85为合格，85-100为优秀。如果成绩不在1-100之间，结果为“错误“。我们使用”rate” 作为评分函数的函数名。

|  |
| --- |
| rate(x) = “不合格” when x >= 0 & x < 60  rate(x < 85) = “合格”  rate(x <= 100) = “优秀”  rate(x) = “错误” |

在上面的示例中，rate是一个函数名。它拥有4种形式（后面会介绍到，这样的情况叫做重载），并且分别在每种形式的rate函数的参数上设置了x<85，x<=100 和when x>=0 & x<60等条件。这样在调用rate函数的时候，就可以根据实际参数的值来确定调用哪一个rate函数。而确定了目标之后，开始调用rate函数，可以得到上面相应的“不合格”，“合格”，“优秀”和“错误”四个结果。例如下面的调用会输出相应的结果：

|  |
| --- |
| print(rate(50)) //打印不合格  print(rate(70)) //打印合格  print(rate(100)) //打印优秀 |

如果你能轻易地看懂上面的例子，那么恭喜你，你可以直接使用WorkScript解决你的工作需要了。但是，为了严谨，我们还是需要理论地介绍函数的每一个特性。当你在编写程序的过程中遇到任何问题，你可以翻阅这本书来解决你的疑惑。

## 4.2 函数的组成

### 4.2.1 函数名

函数名的本质其实是一个变量，当我们声明一个带函数名的函数时，其实相当于创建了这个名字的一个变量，其存储的值为我们的函数。例如上面的rate函数，其中rate就是一个变量，rate变量中所存储的值为上面的四个函数。作为变量，我们也可以对其进行赋值和取值等所有变量所具有操作，尽管我们一般从来不会对函数名这样做。

对于一个函数来说，函数名并不是必须的。当我们声明的函数没有名字时候，就代表我们不需要上述的变量来存储这个函数，而是仅仅声明了一个函数类型的值而已。我们在后面的“匿名函数“小节中会对此进行详细介绍。

### 4.2.2 函数的参数

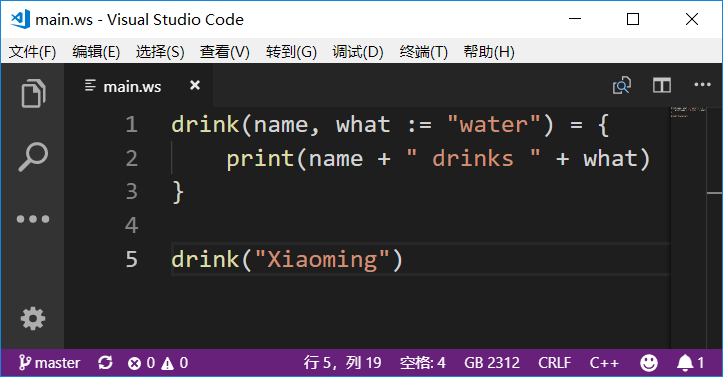
#### 4.2.2.1 参数的声明

分析上面的例子，我们发现rate函数均接收一个参数x。在WorkScript中，一个函数可以有0个到若干个参数，参数必须写在函数名后面的圆括号中。即使一个函数没有参数，圆括号也不可以省略。如果有多个参数，每个参数之间以逗号分隔。逗号不可以省略。

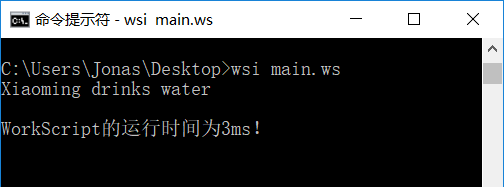
|  |
| --- |
| f() = x //没有参数的情况  f(a) = x //有一个参数的情况  f(a,b,c) = x //有多个参数的情况 |

#### 4.2.2.2 参数的默认值 [暂停提供]

在函数声明中，可以对每个参数设定默认值。当参数设定了默认值时，如果调用时未提供该参数的值，则该参数取默认值。对参数指定默认值，需要使用赋值符号”:=”来指定。下面让我们以下面的例子来理解参数的默认值：



运行结果：

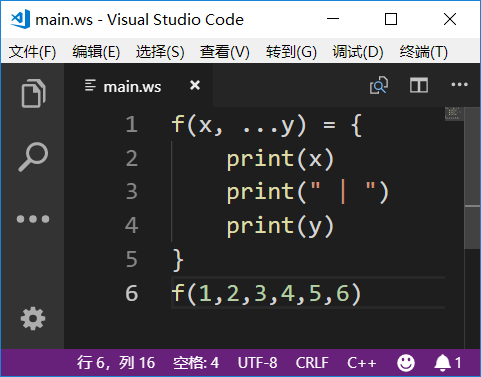
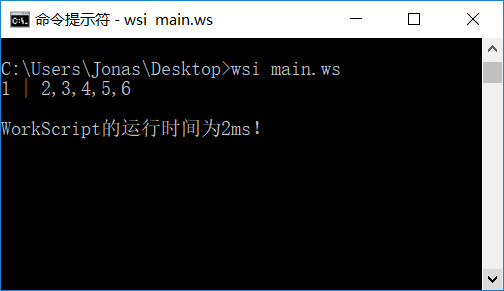


在这个例子中，我们声明了一个drink函数，其拥有两个参数，name和what。并且对what指定了默认值”water”。上例中，在调用drink函数时，我们只传入了第一个参数”Xiaoming”，并没有传入第二个what参数。故what参数遵循默认值的设定，自动赋值为”water”。

#### 4.2.2.3 可变参数 [暂停提供]

之前我们提到，我们可以在最后一个参数之前增加省略号“…”来声明可变参数。如果函数的最后一个参数是可变参数，那么这个参数可以接收多个值。

示例如下：

根据上例，x是第一个参数，接收了值1，而y作为可变参数，接收了2,3,4,5,6。这就是变参函数的使用。

### 4.2.3 函数的约束

上面的rate函数中，除了最后一个声明之外，前三个声明都增加了约束条件。WorkScript的函数的约束条件可以用”when”子句来实现。”when子句”可以写在函数名之前，函数参数列表和函数实现中间，以及函数实现之后。when子句的形式如下：

|  |
| --- |
| when x=1 f(x) = x + 1 //when在函数名之前  f(x) when x=1 = x + 1 //when在中间(如果在合适的地方加括号，看起来会舒服许多。)  f(x) = x + 1 when x=1 //when在最后 |

不管when子句在哪里，如果when子句中包含多行语句，我们都需要增加大括号来包含多条语句（一条语句也可以增加大括号，但显得有些冗余）。以when在最后的形式来举例：

|  |
| --- |
| f(x,y) = x + y  when{  x > 0  y > 0  } |

要注意的是，when中的两条约束的任意一行计算为false，都会使得匹配失败。

另外，大家也注意到了。rate函数的例子中，只有第一条使用了when子句，而第二和第三两条直接在参数列表里写做了“x <= 100“这样的约束。其实，这些属于WorkScript为了方便而设计的语法糖。你可以直接在函数的参数列表中填写大于，小于等约束条件，编译器会自动转换成相应的when子句。让我们来看一下这些约束的简单写法，以及他们会在运行的时候自动被转换成什么样的when子句：

|  |
| --- |
| **//第一种，比较约束：**  f(x <= 100) = x //转换前  f(x) = x when x<=100 //转换后  注意：比较约束的表达式左部必须是参数名。右部可以是任意表达式  上述x <= 100不能写作100 >= x  **//第二种，值约束：**  f(5) = 6 //转换前  f(\_0) = 6 when \_0=5 //转换后（注意，\_0虽然看起来比较奇怪，但也是合法的变量名）  **//第三种：类型约束**  f(number x) = x + 1 //转换前  f(x) = x + 1 when typeof(x)=number //转换后  **//第四种，参数名约束**  equal(x, x) = true //转换前  equal(x, y) = false  equal(\_0, \_1) = true when \_0=\_1 //转换后  equal(x, y) = false |

当然，还有一种特殊的约束，不会被转换为when子句，它就是返回值的类型约束：

|  |
| --- |
| number f(x) = x+1 |

注意，这个number的类型约束是加在函数f上的。它的含义是，函数f的计算结果一定是数字类型。

最后需要强调的是，当一组参数符合一个函数的多种约束的情况下，书写靠前的约束优先匹配。例如评分的函数rate拥有4个声明，那么如果输入的参数为90的话，显然第三条和第四条声明都可以匹配成功。在这时WorkScript约定第三条匹配成功，而不会匹配第四条。

### 4.2.4 函数的实现

上面的rate函数中“合格“，”不合格“，”优秀“和”错误“都叫做函数的实现。只不过上面的例子比较特殊，只有一个值。这样只有一个值的实现，这个值就被当作了返回值（下一小结即将介绍）。一般情况下，函数的实现和约束一样，也可以有一行或者多行语句，并且对于多行语句，必须要用大括号将其包含。

|  |
| --- |
| f(x) = x + 1 //一行实现，省略大括号  f(x) = {  x + 1 //一行实现，不省略大括号  }  f(x) = { //多行实现，不能省略大括号  y = x  y + 1  } |

值得一提的是，如果一个函数的实现没有省略大括号，那么它可以省略等号。例如带大括号的函数实现，我们可以写作如下形式：

|  |
| --- |
| f(x) {  x + 1  } |

另外，大括号可以选择换行开始或者不换行开始。上面的示例全部使用了不换行的格式，但如果你喜欢将语句写得稀疏一些，也可以考虑下面这两种形式：

|  |
| --- |
| f(x) = //这里的=可以省略  {  x + 1  } |

另外，考虑到一些简短的函数声明如果使用等号，会显得比较丑陋，因此规定：在任何形式的函数声明中，你都可以使用箭头号”=>”来代替等号”=”。例如下面的函数声明同样是合法的：

|  |
| --- |
| f(x) => x+1 |

尤其是匿名函数：

|  |
| --- |
| (x) => x + 1 |

### 4.2.5 函数的返回值

一个函数的返回值（在数学中称作函数值），是函数计算的结果。每个函数会有一个计算结果，这和数学是一样的。

例如rate(x <= 100) = “优秀”，其中“优秀“就是这个函数的返回值。如果一个函数的实现只有一行，那么这一行的计算结果就是返回值。如果函数的实现有多行，那么最后一行的计算结果就被作为返回值。让我们举一个例子来看看多行函数的返回值：

|  |
| --- |
| add1(x) = {  print(x)  x + 1  } |

显而易见，x + 1作为add1函数的最后一行语句，它的计算结果会被作为add1这个函数的返回值。而之前的print(x)也会被执行，从而打印出参数x的值，但计算结果不会影响返回值。如果我们调用print(add(1))，将会输出如下结果：

|  |
| --- |
| print(add(1)) //打印出1 2 |

## 4.3 函数的调用

### 4.3.1 按参数顺序传递参数的函数调用

常规的函数调用就是我们日常见到的函数调用，例如print(1,2,3,4,5)，使用函数名print来对print函数进行调用。提供5个参数，分别按位置传递参数，即第一个参数为1，第二个参数为2… 第五个参数为5。这样的函数调用非常容易理解。

### 4.3.2 按参数名传递参数的函数调用[尚未实现]

函数的每个参数都是有名字的，我们在调用的时候，可以直接指定某个参数的值。例如当我们想要求一个长方体的体积，我们需要知道其长宽高三个参数，于是我们可以声明如下函数：

|  |
| --- |
| volume(length, width, height) = length\*width\*height |

这就是一个普通的函数声明，我们在调用时当然可以按三个参数的顺序依次传递参数。但是，同样我们也可以按照参数的名称来指定个别的或全部参数，比如先指定宽，然后按顺序传递长和高。

|  |
| --- |
| volume(width:=5,10,2) |

这里先指定了width=5，然后按照顺序，剩下的参数分别是长和高，按照顺序传入了10和2两个值。这样调用函数，我们同样可以得到100的计算结果。需要强调的是，对参数的指定，我们必须使用明确的赋值符号:=，因为在这个语境下，如果使用=，将会被视为判等，例如上面的调用，会被认为是名为width的变量是否等于5，计算结果为布尔值true或false。

对于按参数名赋值，相当于对参数提前赋值。按名称赋值不会影响正常的按顺序赋值，但是已经被按名称赋值过的参数不会被按顺序重复赋值。例如，上面的例子中，首先对width参数进行了赋值(5)，那么后面的10和2在赋值时，会先对length赋值10，然后跳过width，继续对height赋值为2。

虽然按参数名赋值不会影响正常的按顺序赋值，但从实践上来说，建议在按顺序赋值之后再按参数名赋值，不要将按参数名赋值和按顺序赋值混在一起，这样非常容易造成错误。

### 4.3.3 匿名函数的调用

函数的本质也是一种值，当声明的函数没有函数名的时候，这个函数就仅仅是一个函数类型的值，我们将这种函数称为匿名函数。对于匿名函数性质，在下一小结有更加详细的讨论，这里仅讲解对匿名函数的调用。对于匿名函数的调用，和具名函数的调用规则是一样的。从实践上来说，当我们使用匿名函数的时候，往往是临时使用一次，而不需要将其赋值到变量里。那么，我们可以这样直接进行调用（具名函数同样可以）：

|  |
| --- |
| ((x)=>print(x))(“hello world”) |

这和下面的代码是等效的：

|  |
| --- |
| f = (x)=>print(x)  f(“hello world”) |

## 4.4 函数的性质

### 4.4.1 函数的重载

重载在所有的现代编程语言中几乎都有提供支持，WorkScript也不例外。其实在上一节的rate函数已经使用了重载的特性，而读者一定并没有特别地感觉到上面的例子有什么特别的地方。因为重载这个特性，几乎是一个看起来理所应当的性质。那么什么是重载呢？

***多个函数拥有相同的名称和不同的参数或约束，则互为重载***

注意，重载没有标准的定义，因为各个编程语言的语法千差万别。这个定义是在WorkScript的语法下，最符合重载特性的一种定义了。举例说明，上面的rate函数有四种声明。其实这是四个函数，只不过都叫做rate而已，他们拥有不同的参数列表，从而互相构成了重载。重载的重要作用在于，函数可以会根据运行时的参数来动态地确定匹配哪一个重载。

一般我们从来不会在重载的问题上犯错，除了需要强调，下面这种情况不是重载：

|  |
| --- |
| f(x): number = x + 1  f(x): string = x + ”world” |

因为两个函数拥有相同的参数列表(x)，并且同样地没有任何约束，所以不属于重载。试想，当你调用f(1)的时候，由于两个函数的参数列表完全相同，都可以接收1这个参数。那么该匹配哪一个函数呢？

### 4.4.2泛化函数与特化函数

泛化函数与特化函数的概念非常简单：

***对于有参数的函数来说，有约束的函数都是特化函数，没有约束的函数都是泛化函数。如果一个函数没有参数，那么它既是特化函数，也是泛化函数。***

下面这个函数是泛化函数：

|  |
| --- |
| f(x,y,z) = 1 |

而下面这些函数都是特化函数：

|  |
| --- |
| f(0,y,z) = 1  f(x,y,z>10) = 1  f(x:number, y, z) = 1 |

对于特化函数，不要求必须存在对应的泛化函数。例如下面的特化函数可以单独存在：

|  |
| --- |
| f(0) = 1 |

而不要求必须存在对应的泛化函数：

|  |
| --- |
| f(x) = 1 |

### 4.4.9 外部函数

一个孤独的人是做不成大事的，一门孤立的语言也是难以成器的。我们在WorkScript程序中可以调用其他语言如C/C++编写的函数。使用其他语言编写的函数就叫做外部函数。想要使用外部函数，我们必须对外部函数做声明。外部函数和普通的WorkScript函数很类似，但是有一些特殊的要求：

***外部函数必须明确指定函数名，返回值类型和所有参数的类型***

***外部函数不可以有任何约束和实现部分***

例如下面的f函数是一个合法的外部函数声明：

|  |
| --- |
| int printf(char \*x, ...) |

需要注意的是，外部函数可以和WorkScript函数构成重载。

例如下面的程序是合法的：

|  |
| --- |
| printf(“hello”) = printf(“world”)  int printf(char \*x, ...) |