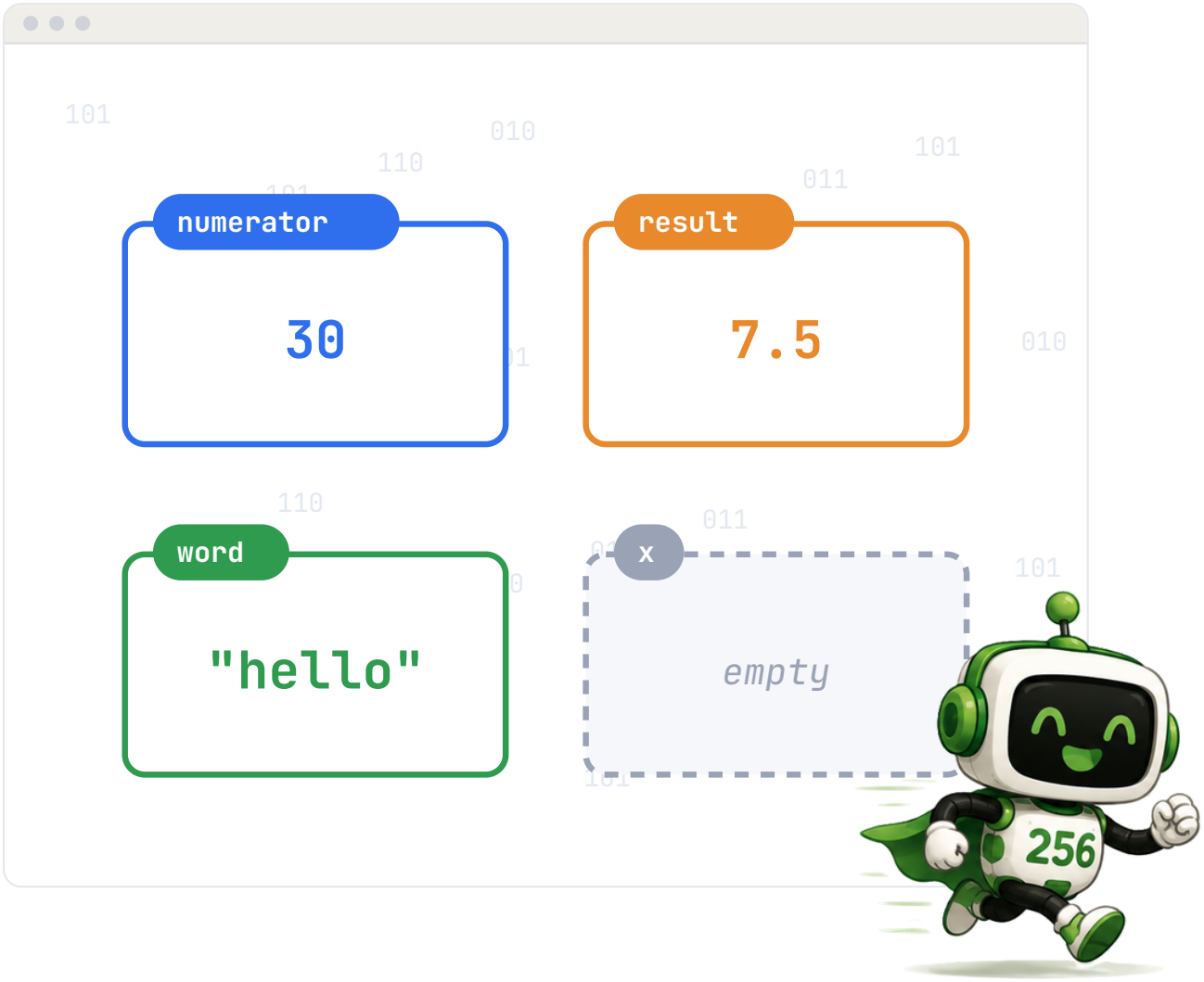
LEARN TO CODE · DATA

Chapter 3

Variables

Programs need somewhere to keep the numbers and words they work with. In this chapter you will give those values names, change them while the program runs, and learn the four kinds of value BASIC-256 understands — and how to turn one kind into another.



© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 1

What is a variable?

Before writing programs that remember things, let’s look at what a variable actually is.

In a computer program, a **variable** is a name for a place in the computer’s memory where a value is kept. Think of it as a labelled box: the label is the variable’s name, and whatever is inside is its value. While the program runs you can put a value in the box, look at it, or swap it for a new one — always by using the name.

In BASIC-256 a variable can hold a whole number (an **integer**), a number with a decimal point (a **floating-point** number) or text (a **string**).



Three variables. The tag is the name; the colour shows the kind of value inside — more on that in Section 4.



NEW CONCEPT

A **variable** gives a name to a piece of storage in the computer’s short-term memory. You can store, change and read back its value anywhere in your program.

A name must **start with a letter**, and after that may use letters, digits, the underscore `_` and the dollar sign `$`. You can’t use a word BASIC-256 keeps for itself, such as `print` or `color` (see Appendix I).

Valid names include `a`, `b6`, `reader`, `f$4`, `slices_each` and `zoo`.

Watch out: variable names are **not** case-sensitive. `score`, `Score` and `SCORE` are three ways to write the *same* name, so they all point to the same box. Pick one spelling and stick to it — your programs will be easier to read.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 2

Assigning values to variables

To put a value into a variable we use assignment. In this introduction we'll use the optional `let` statement.

letsimple.kbs BASIC-256

```
# letsimple.kbs - use variables to store numbers

let numerator = 30
let denominator = 4
let result = numerator / denominator

print numerator + " / " + denominator + " is " + result

let result = result * 2

print "result doubled is " + result
```

TEXT OUTPUT

```
30 / 4 is 7.5
result doubled is 15.0
```

Program 18 · Use Variables to Store Numbers

This program uses three variables. First it stores 30 in the box named `numerator` and 4 in `denominator`. The third `let` takes the value from `numerator`, divides it by the value in `denominator` and stores the answer, 7.5, in `result`.

Look closely at `let result = result * 2`. It reads the value already in `result`, multiplies it by two and puts the answer back into the same variable. The old 7.5 is replaced by 15.0.

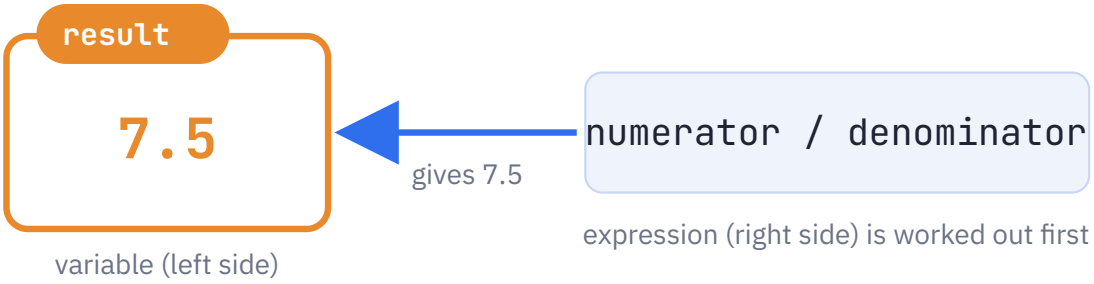
© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



NEW CONCEPT

```
let variable = expression
variable = expression
```

The **let** statement works out the expression on the right of the equals sign, then stores the answer in the variable on the left. This is called **assignment**. The word `let` is optional: `result = 7.5` does exactly the same thing.



Good to know: dividing with `/` always gives a floating-point answer, even when it comes out even. That's why the program prints **15.0** and not 15: `result` came from a division, so it stays a floating-point number when it is doubled. `print 10 / 2` shows `5.0`.

Variables aren't only for numbers. The next program stores two words in variables and then joins them into a sentence with `+`.

letstring.kbs BASIC-256

```
# letstring.kbs - assign a variable a string

let word = "hello"
let rhyme = "yellow"

stuff = word + " and " + rhyme + " are words that rhyme."
print stuff
say stuff
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

TEXT OUTPUT

```
hello and yellow are words that rhyme.
```

says **“hello and yellow are words that rhyme.”**

Program 19 · Use Variables to Store Strings

When one or both sides of a `+` are strings, BASIC-256 joins them together end-to-end instead of adding. The joined sentence is stored in `stuff`, printed, and then spoken with `say`.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 3

Variable assignment shortcuts

In programming there is often more than one way to do something. BASIC-256, like most languages, has shortcuts for adding to and subtracting from a variable. You'll see them in the programs of later chapters.

SHORTCUT	WHAT IT DOES
`variable += expression`	Add the expression to the variable
`variable -= expression`	Subtract the expression from the variable
`variable++`	Add one to the variable — the expression uses the old value
`variable--`	Subtract one — the expression uses the new value
`++variable`	Add one to the variable — the expression uses the new value
`--variable`	Subtract one — the expression uses the new value

Program 20 tries each one. The text after a `#` on a line is a remark, so BASIC-256 ignores it — here the remarks show what each line does.

shortcuts.kbs — BASIC-256

```
# shortcuts.kbs - shortcut assignments

a = 9
a += 10
print a        # 9 + 10 is 19

a++            # same as a = a + 1
print a        # 20

print a++      # print a (20), then add 1
print a        # 21

print ++a      # add 1 first, then print a (22)
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

TEXT OUTPUT

```
19
20
20
21
22
```

Program 20 · Variable Shortcuts

The only tricky part is where the `++` goes. Put it **after** the name and the old value is used first; put it **before** and the variable changes first.

`print a++`

1. print **a** as it is now → **20**
2. then add 1 → a is **21**

use the **old** value

`print ++a`

1. add 1 first → a is **22**
2. then print **a** → **22**

use the **new** value

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 4

Variables and data types

Every value in BASIC-256 has a type, and the type decides what you can do with it.

So far we've said BASIC-256 understands numbers and strings. To be exact, a value is always one of four **data types**. Most languages have many more types, but almost all of them have these four.

0	**unassigned** TYPE_UNASSIGNED no value yet	1 **integer** TYPE_INT 42, -7, 0
2	**floating-point** TYPE_FLOAT 7.5, -0.998, 1.456e10	3 **string** TYPE_STRING "candy bar", 'Why not?'

Unassigned

A variable that has never been given a value is **unassigned**. Using one is almost always a mistake — a misspelled name, or a line in the wrong order — so BASIC-256 stops the program and tells you which variable it is.

unassigned.kbs BASIC-256

```
# unassigned.kbs - use a variable before giving it a value
print '/' + x + '/'
```

TEXT OUTPUT

```
ERROR on line 2: Variable x has not been assigned a value.
```

Program 21 · Unassigned Variable

You can change this in **Preferences ▸ User** with the option **Runtime handling of unassigned variables**:

- **Ignore —** carry on quietly and use an empty value: `''` (nothing) for strings or 0 for numbers.
- **Warn —** use the empty value, but print a WARNING line so you know.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

> **Error —** stop the program with an ERROR. This is the **standard setting**, and the best one while you are learning.

Integers

An **integer** is a whole number — positive, negative or zero — with no fractional part. Use integers to count things and to hold exact values: `42`, `-7`, `0`. Section 6 shows how big they can get.

Floating-point numbers

A number with a decimal point is stored as a **floating-point** number (a "float"). You can type a float as a decimal (`1.45`, `-0.998`, `12345.678`) or in scientific notation, using an `e` to mark the power of ten: `3.24e-1` is 0.324 and `1.456e10` is 14560000000.0. Don't put a thousands separator in a number in your program — `12,345` is not one number.

Strings

A **string** is a sequence of characters — letters, digits, spaces, symbols and punctuation — between quotation marks. You may use single quotes (`'`) or double quotes (`"`), but always close a string with the same kind you opened it with. That lets you put the other kind inside: `"candy bar"`, `"Say 'hi' to her for me."` and `'Why not?'`.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 5

Finding the type of a value

The `typeof` function tells you the type of the data stored in a variable, or the type an expression produces.



NEW CONCEPT

```
typeof( expression or variable )
```

Returns the type of an expression's result or of the value in a variable, as a number: **1** for integers, **2** for floating-point numbers and **3** for strings. A variable that has never been assigned has type **0**. You can compare the result with the constants `TYPE_INT`, `TYPE_FLOAT`, `TYPE_STRING` and `TYPE_UNASSIGNED`.

types.kbs BASIC-256

```
# types.kbs - ask BASIC-256 what type a value is
print "integer 67 is type " + typeof(67)
print "floating-point 2.718 is type " + typeof(2.718)
print "string 'abcd' is type " + typeof('abcd')
print "10 / 2 is type " + typeof(10 / 2)

a = 9
print "variable a containing " + a + " is type " + typeof(a)

a = 74.98
print "variable a containing " + a + " is type " + typeof(a)

a = "nine"
print "variable a containing " + a + " is type " + typeof(a)
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

TEXT OUTPUT

```
integer 67 is type 1
floating-point 2.718 is type 2
string 'abcd' is type 3
10 / 2 is type 2
variable a containing 9 is type 1
variable a containing 74.98 is type 2
variable a containing nine is type 3
```

Program 22 · Data Types

Notice that the same variable `a` holds an integer, then a float, then a string. In BASIC-256 a variable takes on the type of whatever was last stored in it. And as promised, `10 / 2` is type 2: division always hands back a float.



EXPLORE

With the standard settings, `typeof(a)` on a variable that was never assigned stops the program, just like Program 21. Switch **Runtime handling of unassigned variables** to **Warn** and add `print typeof(q)` to the program — this time it prints **0** (`TYPE_UNASSIGNED`) after the warning. Switch the setting back to Error when you are done.

Coming later: `typeof` can also answer **4** (array), **5** (reference) and **6** (map), with the constants `TYPE_ARRAY` , `TYPE_REF` and `TYPE_MAP` . You'll meet those in later chapters.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

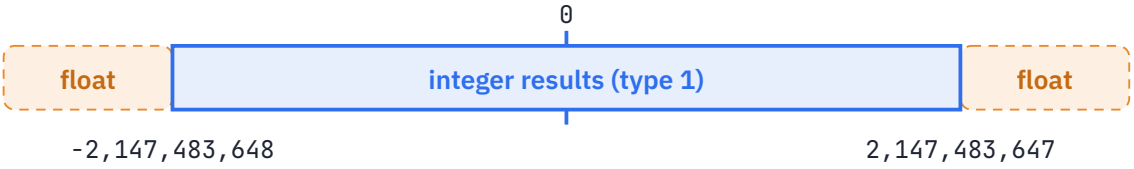
SECTION 6

Numbers up close

Integers and floats are both numbers, but they are stored differently — and each has its limits.

How big can an integer be?

When you calculate with integers, the answer stays an integer as long as it fits between **−2,147,483,648** and **2,147,483,647**. If an answer goes past either end, BASIC-256 doesn't give you a wrong number — it quietly hands the answer back as a floating-point number instead.



A result that falls outside the blue range is handed back as a floating-point number (type 2).

How big can a float be?

Floating-point numbers can be enormous or tiny: roughly from 2.2×10^{-308} up to 1.8×10^{308}, positive or negative. The price is precision: a float is only an *approximation*, good to about 15 significant digits. A calculation that goes past the top of the range stops the program with *Operation returned infinity*.

bignumbers.kbs BASIC-256

```
# bignumbers.kbs - what happens at the edges of the number types

big = 2147483647
print big + " is type " + typeof(big)

big += 1
print big + " is type " + typeof(big)

print 100000 * 100000
print 12345678901234.5
print 1 / 3
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

TEXT OUTPUT

```
2147483647 is type 1
2147483648.0 is type 2
10000000000.0
1.23456789012e+13
0.33333333333
```

Program 23 · Numbers at the Edges

How floats are printed

When BASIC-256 prints a float it shows up to **12 significant digits**, and always includes a decimal point — adding `.0` if needed — so you can tell a float from an integer at a glance. Very large or very small numbers switch to scientific notation. No thousands separators are added.

YOU WRITE	BASIC-256 PRINTS
`print 7.5`	`7.5`
`print 15.0`	`15.0`
`print 1234567890.0`	`1234567890.0`
`print 12345678901234.5`	`1.23456789012e+13`
`print 1 / 3`	`0.33333333333`
`print 3.24e-1`	`0.324`
`print 1.456e10`	`14560000000.0`

Every result above was printed by BASIC-256 with its standard settings.



BEHIND THE SCENES

Inside, BASIC-256 keeps every integer in 64 bits (8 bytes) of memory and every float in the standard 64-bit “double” format. The 12-digit display is just a setting: **Preferences ▸ User** lets you show anywhere from 10 to 16 digits, choose whether floats get the trailing `.0`, and use your country’s decimal separator (a comma in much of Europe).

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 7

Converting values from one type to another

BASIC-256 has three functions that turn a value of one type into another: `int()`, `float()` and `string()`.



NEW CONCEPT

```
int( expression )
```

Returns an integer.

If the expression is a floating-point number, the part after the decimal point is cut off and only the whole part is returned. There is **no rounding**: `int(4.9)` is 4 and `int(-3.7)` is −3.

If the expression is a string, BASIC-256 tries to read a number from it: `int("42")` is 42 and `int("3.9")` is 3. If the string isn't a number, you get **0**.



NEW CONCEPT

```
float( expression )
```

Returns a floating-point number. An integer comes back with the same value as a float (`float(3)` is 3.0). A string is read as a number (`float("2.5")` is 2.5); if the string isn't a number, you get **0**.



NEW CONCEPT

```
string( expression )
```

Returns a string. If the expression is a number (integer or float), you get a string containing that number, written just the way `print` would show it.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

Watch out: by default a string that isn't a number — like `int("abc")` — turns into 0 **without any message**. If you'd rather be told, set **Runtime handling of bad type conversions** in **Preferences ▸ User** to Warn or Error.

intandstring.kbs BASIC-256

```
# intandstring.kbs - convert between types

a = 9 / 2

# convert a to a string and concatenate
print "a is " + string(a)

# convert a to an integer
print "int(a) is " + int(a)

# round a to the nearest integer
print "a rounded is " + int(a + .5)
```

TEXT OUTPUT

```
a is 4.5
int(a) is 4
a rounded is 5
```

Program 24 · Converting Data Types

The last line uses a classic trick: adding 0.5 before cutting off the decimals rounds to the nearest whole number. 4.5 + 0.5 is 5.0, and `int` keeps the 5.



TRY IT

The +0.5 trick only works for positive numbers — `int(-2.5 + .5)` gives −2. BASIC-256 also has a `round()` function that handles both: `round(-2.5)` is −3.0. Notice the `.0`: `round` hands back a float.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



BIG PROGRAM

The Pizza Party Planner

Every chapter ends with a bigger program to type in and play with, using only what you've learned so far. This one works out how many pizzas to order for a party — and shows integers, floats, rounding and string joining all working together.

pizzaparty.kbs — BASIC-256

```
# pizzaparty.kbs - plan the pizza for a party

guests = 13
slices_each = 3
slices_per_pizza = 8
price = 11.5

slices_needed = guests * slices_each
print "Slices needed: " + slices_needed

exact = slices_needed / slices_per_pizza
print "Exact number of pizzas: " + exact

# round UP: add one pizza less a slice, then cut off the fraction
pizzas = int((slices_needed + slices_per_pizza - 1) / slices_per_pizza)
print "Pizzas to order: " + pizzas

leftover = pizzas * slices_per_pizza - slices_needed
print "Slices left over: " + leftover

total = pizzas * price
print "Total cost: " + total

each = total / guests
print "Cost per guest: " + each

# keep two decimals: shift, round, shift back
each = int(each * 100 + 0.5) / 100
print "Rounded cost per guest: " + each

say "Order " + pizzas + " pizzas"
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

TEXT OUTPUT

```
Slices needed: 39
Exact number of pizzas: 4.875
Pizzas to order: 5
Slices left over: 1
Total cost: 57.5
Cost per guest: 4.42307692308
Rounded cost per guest: 4.42
```

says **“Order 5 pizzas”**

Program 25 · Big Program — Pizza Party Planner

The trickiest line is the one that rounds *up*. Adding one pizza’s worth of slices, less one, before dividing pushes any partial pizza over the next whole number; `int` then cuts off the fraction. 39 slices + 7 is 46, and 46 ÷ 8 is 5.75 — so `int` gives 5 pizzas. Change `guests` and `price` and run it again.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



PRACTICE

Exercises

WORD SEARCH

u j y s t r i n g t z l
p n p s e d q f n y e z
o k a c h q v e l l f f
y v e s e o m k b o j d
w q e t s n r a j e a g
e r x r g i i t b y f t
k t z i f r g v c o r x
w g s f a l j n e u n r
f s b v w v o p e h t i
a y c v o z y w n d r i
i n t r o t r o a l e t
i n t e g e r m k f i p

Find: assignment, float, int, integer, let, overflow, shortcut, string, typeof, unassigned, variable



Problems

Try these — they only use what this chapter covered.

1 Create a program with two variables `a` and `b` and give each a number. Print the sum of a and b, the difference of a and b, the difference of b and a, the product of a and b, the quotient of a divided by b, and the quotient of b divided by a. Run it with several values. Which answers are integers and which are floats? What happens when a or b is zero?

2 Store a temperature in Celsius in a variable called `celsius`. Work out the Fahrenheit temperature with `celsius * 9 / 5 + 32`, store it in `fahrenheit`, and print a sentence such as *37 C is 98.6 F*. Try 37 and 100 — why does one answer end in `.0`?

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

3 Without running it, write down what this program prints. Then type it in and check.

```
n = 5
n += 3
print n++
print n
print --n
print typeof(n / 2)
```

4 Put 5 in `x` and 8 in `y`. Write lines that **swap** them, so x ends up 8 and y ends up 5, then print both. Hint: you'll need a third variable to hold one value for a moment.

5 Set `big = 2147483640`. Use `big += 5` a few times, printing `big` and `typeof(big)` after each one. On which line does the type change, and why?

Chapter 3 · Variables. Modernised layout of the BASIC-256 beginner's guide, updated for BASIC-256 2.3: names are not case-sensitive, unassigned variables stop the program by default, integer results that overflow become floats, and floats print without thousands separators. Every sample output in this chapter was produced by running the program in BASIC-256 2.3. Original text © 2019 James M. Reneau, licensed under CC BY-NC-SA 3.0 US. This adaptation is shared under the same licence. Mascot: BitBot.