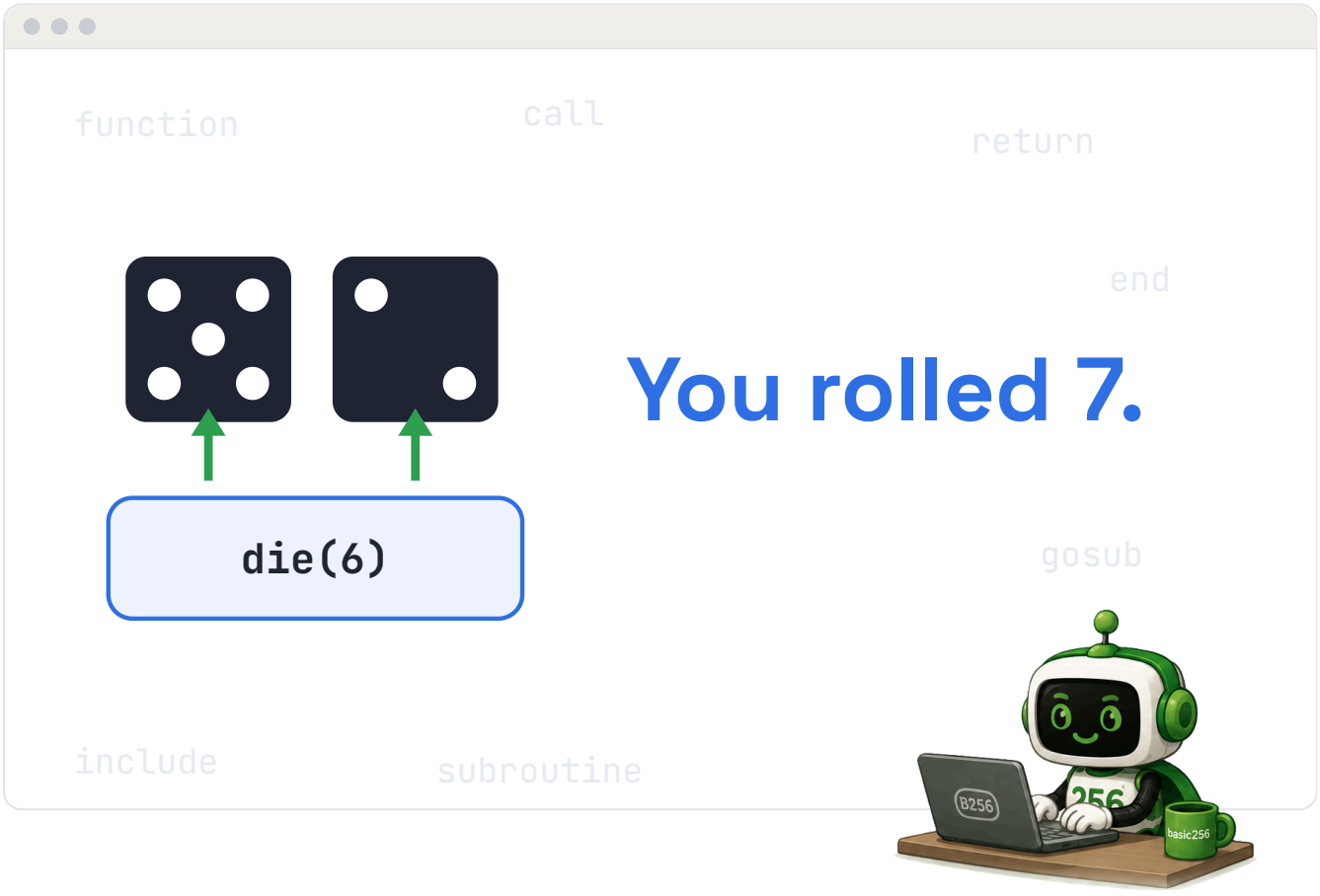
LEARN TO CODE · REUSING CODE

Chapter 10

Functions and Subroutines — Reusing Code

Programs grow. In this chapter you learn to give a piece of code a name and use it again and again: **functions** that work out a value, and **subroutines** that do a job. You put them in files of their own to share them between programs, and you meet the older way of jumping around a program with labels, **goto** and **gosub**.



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

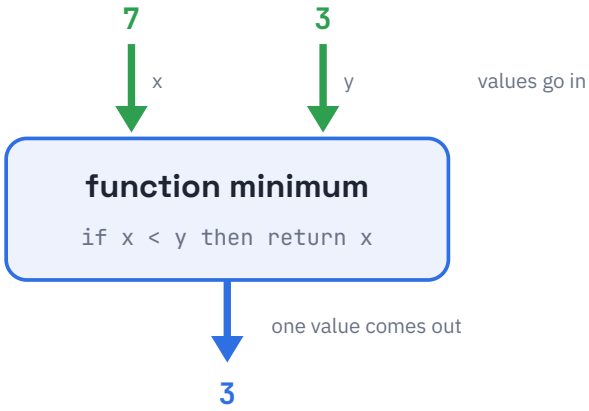
SECTION 1

Functions

A function is a small program inside your program: you give it values, and it gives you one value back.

Programmers write functions and subroutines to test small parts of a program on their own, to use the same code wherever it is needed, to add new words to the language, and to make long programs easier to read. You already know some functions that come with BASIC-256, like **rand**, **int** and **rgb**. Now you will write your own.

You can send a function zero or more values, called **arguments**. The function works with them and returns one value.



A function. Here two values go in, 7 and 3, and the smaller one comes out.

NEW CONCEPT

```
function name(argument(s))
   statements
end function
```



function starts a block of statements and gives it a name. The brackets after the name are required, even when the function takes no arguments. In them you list the variables that receive the values sent by the part of the program that *calls* the function. Those variables belong to the function.

end function (or **endfunction**) closes the definition. A function may not be defined inside a loop, an **if** or another function: put your functions after the main program.

The function hands back its value with **return**, or by setting a variable with the function's own name. If it does neither, it gives back 0. Don't give a function the same name as one of your variables: the program still runs, but it gets confusing.

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



NEW CONCEPT

```
return value
```

In a function, **return** stops the function straight away and sends the value back to where the function was called. The statements after it are not run.

minimum.kbs BASIC-256

```
# minimum.kbs - a function that finds the smaller number

inputfloat "Enter a number: ", a
inputfloat "Enter a second number: ", b

print "The smaller one is " + minimum(a, b)
end

function minimum(x, y)
   # return the smaller of the two numbers
   if x < y then return x
   return y
end function
```

Program 64 · Minimum Function

TEXT OUTPUT

```
Enter a number: 7
Enter a second number: 3
The smaller one is 3.0
```

The main program ends at **end**. Below it is the function. When line 6 needs `minimum(a, b)`, the value of *a* goes into *x* and the value of *b* into *y*. If *x* is the smaller one, the function returns it at once; otherwise it carries on to the last **return** and gives back *y*.



NEW CONCEPT

```
end
```

end stops the program. It is a good habit to put it at the end of the main program, before your functions and subroutines, so that you can see where the main program stops. (BASIC-256 skips function and subroutine definitions anyway, so here it is not strictly needed. In Section 4 you will meet a case where it is.)

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

Program 65 has two functions. **getinteger** asks for a whole number with a default answer, and **die** rolls a die with any number of sides.

gameroller.kbs BASIC-256

```
# gameroller.kbs - roll some dice

print "Dice roller"
s = getinteger("Sides on each die", 6)
n = getinteger("Number of dice", 2)
total = 0
for x = 1 to n
   d = die(s)
   print d
   total += d
next x
print "Total " + total
end

function getinteger(message, default)
   # ask for a whole number - if the answer is empty,
   # or not a whole number, use the default instead
   input message + " (default " + default + ")? ", n
   if typeof(n) <> 1 then n = default
   return n
end function

function die(sides)
   # roll one die: a number from 1 to sides
   return int(rand * sides) + 1
end function
```

Program 65 · Dice Roller

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

TEXT OUTPUT

```
Dice roller
Sides on each die (default 6)? 6
Number of dice (default 2)? 3
4
1
4
Total 9
```

```
Dice roller
Sides on each die (default 6)?
Number of dice (default 2)?
1
4
Total 5
```

In the second run the user only pressed Enter, twice, so the defaults were used: two dice with six sides.

getinteger uses plain **input** and **typeof** from Chapter 6. If you type a whole number, **input** stores it as an integer and **typeof** gives 1. If you just press Enter, or type something like *abc* or 2.5, the type is not 1, and the function returns the default instead.

Variables in a function are local. The main program and **getinteger** both use a variable called *n*, but they are two different variables: setting *n* inside the function doesn't change the *n* of the main program. It works the other way round too: a function can't see the variables of the main program. If it needs a value, pass it as an argument.

A function can return a string just as easily as a number. Program 66 builds a string by adding a word to it again and again with `+=`.

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

repeatstring.kbs BASIC-256

```
# repeatstring.kbs - a function that returns a string

a = "hi"
b = repeat(a, 10)
print a
print b
end

function repeat(word, times)
   result = ""
   for t = 1 to times
      result += word
   next t
   return result
end function
```

TEXT OUTPUT

```
hi
hihihihihihihihihihi
```

Program 66 · Repeating String Function



TRY IT

Add `print repeat("-", 30)` to the main program to draw a line of thirty dashes. Then use **repeat** to print a triangle of stars: one star on the first line, two on the second, up to ten.

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

SECTION 2

Subroutines

A subroutine is like a function, but instead of working out a value it does a job.

A subroutine is a block of code with a name that does something specific, such as drawing a picture or showing the time. Different parts of a program can use the same subroutine, and you can send it values to tell it what exactly to do. Subroutines are like functions, except that they don't return a value, and you start one with the **call** statement.



NEW CONCEPT

```
subroutine name(argument(s))
   statements
end subroutine
```

subroutine starts a block of statements and gives it a name. As with functions, the brackets are required, the variables listed in them receive the values sent by **call**, and all variables in the subroutine are local. **end subroutine** (or **endsubroutine**) closes the definition.



NEW CONCEPT

```
call name(value(s))
```

call runs the subroutine, passing it the values. When the subroutine is finished, the program carries on with the line after the **call**. You can't leave out **call** or the brackets: `displaytime()` or `call displaytime` on its own is a syntax error.



NEW CONCEPT

```
return
```

In a subroutine, **return** without a value leaves the subroutine early. You don't need it at the end: reaching **end subroutine** returns as well.

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

NEW CONCEPT




```
year
month
day
hour
minute
second
```

These functions read the computer's clock and calendar, so a program can tell what time it is. The brackets are optional.

year	The year, with four digits.
month	The month — **counting from 0**: 0 is January, 1 February … 11 December.
day	The day of the month, 1 to 31.
hour	The hour on a 24-hour clock, 0 to 23: 0 is midnight, 13 is 1 in the afternoon.
minute	The minute, 0 to 59.
second	The second, 0 to 59.

Program 67 is a ticking clock. Every second the main loop calls the subroutine **displaytime**, which paints a blue rectangle over the old time and writes the new one. The **while true** loop runs forever, like Program 48 in Chapter 8, so stop the program with the Stop button.

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

subroutineclock.kbs BASIC-256

```
# subroutineclock.kbs - a ticking clock

fastgraphics
font "Tahoma", 40, 700
clg blue
color yellow
text 20, 20, "My Clock"

while true
   call displaytime()
   pause 1
end while
end

subroutine displaytime()
   # paint over the old time
   color blue
   rect 100, 200, 300, 100
   # and write the new time
   color yellow
   t = padtwo(hour) + ":" + padtwo(minute)
   t += ":" + padtwo(second)
   text 130, 220, t
   refresh
end subroutine

function padtwo(x)
   # put a 0 in front of a single digit
   if x < 10 then x = "0" + x
   return x
end function
```

GRAPHICS OUTPUT



Program 67 · Subroutine Clock

The function **padtwo** puts a 0 in front of a number below 10, so that five past nine shows as 09:05 instead of 9:5. **fastgraphics** and **refresh** from Chapter 8 make sure you never see the clock half drawn.

Program 68 shows the date as well. Because **month** counts from 0, the program adds 1 before showing it. Both subroutines are called every second, so the date also changes at midnight, and a single **refresh** in the main loop shows the date and the time together.

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

subroutineclockimproved.kbs BASIC-256

```
# subroutineclockimproved.kbs - date and time

fastgraphics
font "Tahoma", 40, 700
clg blue

while true
   call displaydate()
   call displaytime()
   refresh
   pause 1
end while
end

subroutine displaydate()
   color blue
   rect 50, 140, 400, 90
   color yellow
   # month counts from 0, so add 1
   d = padtwo(day) + "/" + padtwo(month + 1)
   d += "/" + year
   text 85, 155, d
end subroutine

subroutine displaytime()
   color blue
   rect 50, 250, 400, 90
   color yellow
   t = padtwo(hour) + ":" + padtwo(minute)
   t += ":" + padtwo(second)
   text 130, 265, t
end subroutine

function padtwo(x)
   # put a 0 in front of a single digit
   if x < 10 then x = "0" + x
   return x
end function
```

GRAPHICS OUTPUT



Program 68 · Subroutine Clock — Improved

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



TRY IT

The date is written day/month/year. If you write dates differently, change the order in **displaydate** — for example year-month-day, as in 2026-11-03.

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

SECTION 3

Using the same code in several programs

With `include` you keep functions in a file of their own and use them in any program.

Once you have written a good function, you will want it in other programs too. You could copy and paste it — but if you later improve it, you have to change every copy. The **include** statement solves that: when you run the program, BASIC-256 first reads the code from the other file, as if you had typed it in that place.

Program 69 is Program 65 again, but its two functions have been saved as files of their own: Programs 70 and 71.

gamerollerinclude.kbs BASIC-256

```
# gamerollerinclude.kbs - roll dice with included functions

include "diefunction.kbs"
include "getintegerfunction.kbs"

print "Dice roller with included functions"
s = getinteger("Sides on each die", 6)
n = getinteger("Number of dice", 2)
total = 0
for x = 1 to n
   d = die(s)
   print d
   total += d
next x
print "Total " + total
end
```

Program 69 · Dice Roller — With Included Functions

TEXT OUTPUT

```
Dice roller with included functions
Sides on each die (default 6)? 20
Number of dice (default 2)? abc
11
15
Total 26
```

abc is not a whole number, so the default of 2 dice was used.

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

diefunction.kbs BASIC-256

```
# diefunction.kbs - roll a die with any number of sides

function die(sides)
   return int(rand * sides) + 1
end function
```

Program 70 · The die Function

getintegerfunction.kbs BASIC-256

```
# getintegerfunction.kbs - ask for a whole number
# if the answer is empty, or not a whole number,
# use the default instead

function getinteger(message, default)
   input message + " (default " + default + ")? ", n
   if typeof(n) <> 1 then n = default
   return n
end function
```

Program 71 · The getinteger Function



NEW CONCEPT

```
include "file name"
```

include adds the code from another file when the program is run. The file name must be written in quotes: it can't be a variable or another expression. Save the files you include in the same folder as the program that includes them.

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



EXPLORE

BASIC-256 also comes with some ready-made files of functions and subroutines, called **modules**: *math.kbs* with extra mathematics, *turtle.kbs* for turtle graphics, and *toolkit.kbs* with buttons, sliders and other controls on the graphics output. You include a module just like your own files, for example `include "math.kbs"`, and BASIC-256 knows where to find it. The name of every function and subroutine in a module starts with the module's two letters and an underscore — `ma_`, `tu_` or `tk_` — so they never clash with your own names. To see what is in a module, include it and call its help subroutine, which prints a list of everything in it:

```
include "math.kbs"
call ma_help()
```

For the other two modules, call `tu_help()` or `tk_help()`. The modules are covered in the Advanced chapters.

Now that the functions are in files of their own, other programs can use them without changing or retyping them. Program 72 uses **getinteger** to make an adding machine.

addingmachine.kbs BASIC-256

```
# addingmachine.kbs - a simple adding machine

include "getintegerfunction.kbs"

print "Adding machine"
print "Press Stop to end"

total = 0
while true
   a = getinteger("+", 0)
   total += a
   print total
end while
```

Program 72 · Adding Machine

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

TEXT OUTPUT

```
Adding machine
Press Stop to end
+ (default 0)? 6
6
+ (default 0)?
6
+ (default 0)? 55
61
+ (default 0)? 2.5
61
+ (default 0)?
```

Pressing Enter adds the default, 0, and 2.5 is not a whole number, so it counts as 0 too.

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

SECTION 4

Labels, goto and gosub

The old way of jumping around a program, from before BASIC had functions and subroutines.



THINK ABOUT IT

This section shows how programs used to jump around before subroutines and functions were added to the language. You will still see it in old programs. But **goto** and **gosub** easily make programs that are tangled and hard to follow, so in your own programs use loops, functions and subroutines instead.

In Chapter 8, Program 48 looped forever with **while true**. You can do the same with a label and a **goto** statement.

goto.kbs BASIC-256

```
# goto.kbs - loop forever with a label
top:
print "hi"
goto top
```

TEXT OUTPUT

```
hi
hi
hi
hi
… forever
```

Program 73 · Goto with a Label



NEW CONCEPT

```
label:
```

A **label** gives a place in your program a name, so that you can jump to it. It is a name followed by a colon, at the start of a line (spaces in front are allowed), and a statement may follow it on the same line. A label name starts with a letter and may contain letters, digits and underscores. It can't be a BASIC-256 word such as *print*, and each label may appear only once. Capitals don't matter: `goto TOP` finds `top:`.

Examples of labels: `top:`, `far999:` and `about_box:`.



NEW CONCEPT

```
goto label
```

goto makes the program jump to the label and carry on with the statement after it.

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

Like subroutines and functions, **gosub** lets you use a block of code several times. The big difference is that a **gosub** block has no variables of its own: every variable belongs to the whole program. Program 74 jumps to the code at the label *showline* three times, and that code changes the variable *a* of the main program.

gosub.kbs BASIC-256

```
# gosub.kbs - a simple gosub

a = 10
for t = 1 to 3
   print "a equals " + a
   gosub showline
next t
end

showline:
print "------------------"
a = a * 2
return
```

TEXT OUTPUT

```
a equals 10
------------------
a equals 20
------------------
a equals 40
------------------
```

Program 74 · Gosub



NEW CONCEPT

```
gosub label
return
```

gosub jumps to the label, like **goto**, but remembers where it came from. A **return** jumps back to the statement after the **gosub**.



TRY IT

Take the **end** out of Program 74 and run it. After the loop, the program runs straight on into the lines after the label: it prints one more line of dashes, and then the **return** has nowhere to go back to. BASIC-256 stops with *ERROR on line 12: Unexpected RETURN.* With **gosub**, the **end** is needed.

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



BIG PROGRAM

Roll Two Dice

Let's roll two dice, draw them on the graphics output and give the total. The program includes the **die** function from Program 70 to roll, and uses a subroutine to draw a die, so the drawing code is written only once.

rollgraphicaldice.kbs BASIC-256

```
# rollgraphicaldice.kbs - roll two dice and draw them

include "diefunction.kbs"

clg
total = 0

roll = die(6)
total += roll
call drawdie(60, 175, roll)

roll = die(6)
total += roll
call drawdie(290, 175, roll)

print "You rolled " + total + "."
end

subroutine drawdie(x, y, n)
   # draw a 150 x 150 die, top left corner at x, y,
   # with n spots: circles with a radius of 15
   color black
   rect x, y, 150, 150
   color white
   # top row
   if n <> 1 then circle x + 35, y + 35, 15
   if n = 6 then circle x + 75, y + 35, 15
   if n >= 4 then circle x + 115, y + 35, 15
   # middle
   if n = 1 or n = 3 or n = 5 then circle x + 75, y + 75, 15
   # bottom row
   if n >= 4 then circle x + 35, y + 115, 15
   if n = 6 then circle x + 75, y + 115, 15
   if n <> 1 then circle x + 115, y + 115, 15
end subroutine
```

GRAPHICS OUTPUT



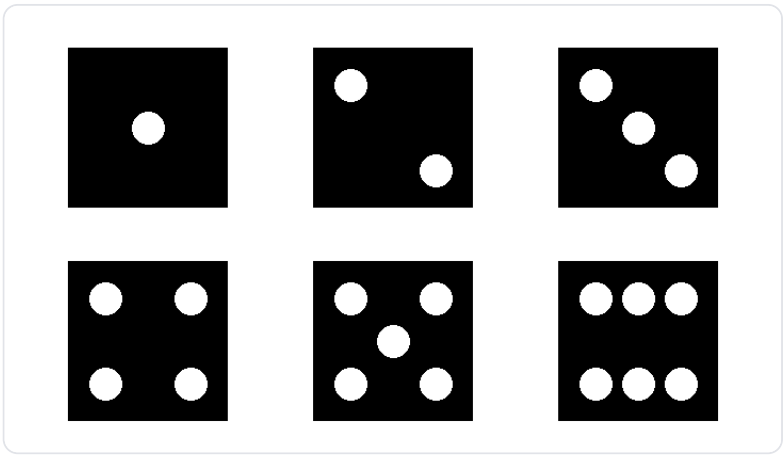
Program 75 · Big Program — Roll Two Dice

Each spot is a white circle on the black square. A spot is drawn only for the numbers that have it: the top left one for every number except 1, the middle one for the odd numbers, and so on.

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

TEXT OUTPUT

```
You rolled 7.
```



All six faces, drawn by calling **drawdie** six times.



TRY IT

Add a third die. Then make the program keep rolling until both dice show the same number: put the rolling and drawing in a loop with a **pause**, and count the rolls.

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



PRACTICE

Exercises

WORD SEARCH

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

Find: argument, call, day, end, file, function, gosub, goto, hour, include, label, minute, month, parenthesis, return, second, subroutine, terminate, year

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

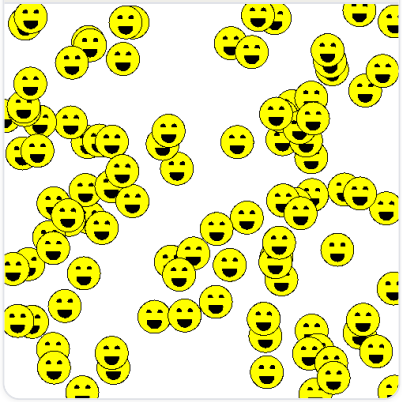


Problems

Try these — they only use what this chapter and the earlier ones covered.

1 Write a subroutine that takes two numbers: a point on the graphics output. Have it draw a smiling face with a radius of 20 pixels at that point, using circles, rectangles or polygons. Call the subroutine 100 times in a loop, at random places, to fill the screen with smiles.

GRAPHICS OUTPUT



2 Write a program that asks for two points, *x1*, *y1* and *x2*, *y2*, and shows the formula of the straight line through them as $y = mx + b$. Write a function that returns the slope $m = (y1 - y2) / (x1 - x2)$, and a second function that returns $b = y - m \times x$ when you give it one of the points and the slope.

TEXT OUTPUT

```
x1? 1
y1? 1
x2? 3
y2? 2
y = 0.5x + 0.5
```

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

3 In mathematics, *n*! (say "*n* factorial") means 1 × 2 × 3 × … × *n*, and 0! is 1. Write a function that takes one number and returns its factorial. Call it in a **for** loop to show 1! to 10!.

TEXT OUTPUT

```
1! is 1
2! is 2
3! is 6
4! is 24
5! is 120
6! is 720
7! is 5040
8! is 40320
9! is 362880
10! is 3628800
```

4 A **recursive** function is a function that calls itself. Knowing that *n*! = *n* × (*n* − 1)! and that 0! = 1, rewrite problem 3 with a recursive factorial function. (Each call has its own local variables, so this works.)

5 Change the Big Program so that the subroutine takes a fourth argument: the colour of the spots. Roll the dice ten times, half a second apart, with spots in a random colour each time.

Chapter 10 · Functions and Subroutines — Reusing Code. Modernised layout of the BASIC-256 beginner's guide, updated for BASIC-256 2.3. Programs are numbered 64–75, continuing from Chapter 9. Corrected for 2.3: `month` counts from 0, so the improved clock now adds 1 (it showed the previous month); labels are not case-sensitive and may be indented; `call` and the brackets can't be left out; a function without `return` gives 0; a function can't see the main program's variables; `end` is needed before `gosub` code but not before functions; string functions need no `$`. Also made consistent: the dice roller's function is called `getinteger` in all programs, typeof is compared with 1 everywhere, `padtwo` is used in both clocks, and `+=` replaces the untaught `;=`. The clocks use a bold font (weight 700), are redrawn for the 500 × 500 graphics output, refresh once per second and update the date too; the dice are bigger and their spots are circles. New: a pointer to the modules that come with BASIC-256. Every output 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.