Beginner's Guide to CODING

Discover the joy and art of computer programming with your Raspberry Pi

Learning to code is one of the most profoundly life-changing things you can do. This has always been true, but learning to code is increasingly important in the modern world.

The reason the Raspberry Pi was created was to challenge a drop in computer science applications at Cambridge University. Modern computers, and especially games consoles, were fun and powerful, but not easily programmable.

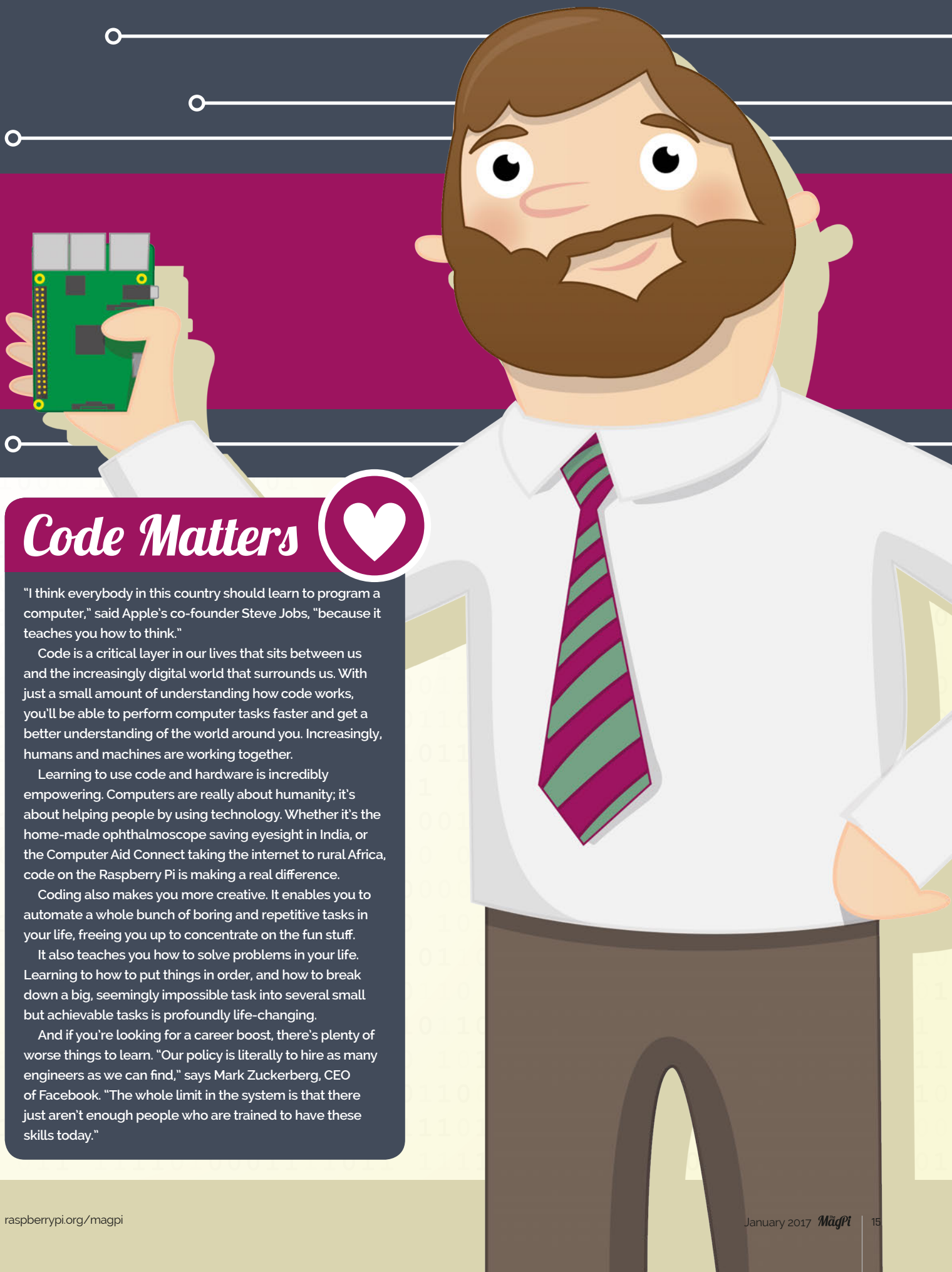
The maker community fell in love with the Raspberry Pi, thanks to its cheap and hackable nature. Building and tinkering are the primary reasons we love Raspberry Pi. Great projects use a combination of hardware and software together.

So, whether you're a hacker learning to make better projects, or a would-be coder looking for a better career, this feature is set to help you on your way.

The good news is that you don't need to be a genius to know coding, just as you don't have to be a genius to read and write. It's actually pretty simple once you learn a few simple concepts like variables, branching, and loops.

Perhaps you're brand-new to coding. Maybe you did a little BASIC in school, or used old languages like Pascal and Fortran. Or maybe you're already knee-deep in projects and just want to learn the language that controls them.

Wherever you're coming from, we're here to walk you through the basic concepts of computer programming. We'll demystify the whole process of code, so you can get a better understanding of what's going on inside your Raspberry Pi.



Code Matters

"I think everybody in this country should learn to program a computer," said Apple's co-founder Steve Jobs, "because it teaches you how to think."

Code is a critical layer in our lives that sits between us and the increasingly digital world that surrounds us. With just a small amount of understanding how code works, you'll be able to perform computer tasks faster and get a better understanding of the world around you. Increasingly, humans and machines are working together.

Learning to use code and hardware is incredibly empowering. Computers are really about humanity; it's about helping people by using technology. Whether it's the home-made ophthalmoscope saving eyesight in India, or the Computer Aid Connect taking the internet to rural Africa, code on the Raspberry Pi is making a real difference.

Coding also makes you more creative. It enables you to automate a whole bunch of boring and repetitive tasks in your life, freeing you up to concentrate on the fun stuff.

It also teaches you how to solve problems in your life. Learning to how to put things in order, and how to break down a big, seemingly impossible task into several small but achievable tasks is profoundly life-changing.

And if you're looking for a career boost, there's plenty of worse things to learn. "Our policy is literally to hire as many engineers as we can find," says Mark Zuckerberg, CEO of Facebook. "The whole limit in the system is that there just aren't enough people who are trained to have these skills today."

What is a Program?

Discover the building blocks of software and learn what goes on inside a program

Which Python?

Python 2 and Python 3 are both commonly used. Python 3 is the future, so we're going with it. Lots of courses still teach Python 2, and it's not a bad idea to take a closer look at the differences between the two: **magpi.cc/2gP6zX3**

Before you go any further, let's look at what a program actually is. The dictionary definition is a "set of instructions that makes a computer do a particular thing."

A computer program is a lot like a recipe. It has a list of ingredients, called 'variables', and a list of instructions, known as 'statements' or 'functions'. You follow the instructions from the recipe one line at a time and end up with a tasty cake – and that's no lie.

The real miracle of computers, however, is that they can do the same thing repeatedly. So you can get a machine to bake a thousand cakes without ever getting tired. A program may contain loops that make it do the same thing over and over again.

Programs also make decisions, and different paths through a program can be taken. Your recipe could make a scrummy chocolate cake or a delightful batch of doughnuts, depending on the variables (the ingredients) it has.

Compiled vs Interpreted

Python is an 'interpreted language'. You write the code and then run the program. Under the hood, it's being translated and runs on the fly. Some languages, such as C and Java, are compiled. You write the program, then compile it to get a build file (written in machine code), then you run the build. It's a faff you can do without for now.

One thing that may surprise you when you begin programming is just how little you need to know to get started. With just a few variables, a smattering of flow, and some functions, you can get a computer doing all the hard work for you.

Inside your Pi

At the heart of your Raspberry Pi are billions of voltage switches known as binary digits (or 'bits' for short). There are 8,589,934,592 of them in its 1GB of RAM, to be exact. All these switches can be set to high or low, which is typically represented as 0 (for low or off) and 1 (for high or on). Everything you see on the screen, hear from the speakers, and type on the keyboard is billions of switches being turned on and off.

Obviously, it's not that easy for humans to talk directly to computers. It's possible to use machine language and send binary instructions directly to a computer, but this isn't where any sane person starts (or ends if they want to remain sane).

Instead, we use a coding language to program. This is written using easy-to-understand functions like `print()`. These are then interpreted into machine language, which the computer understands.

We're going to use Python to learn to code. Python is a truly great programming language. It has a rich syntax that's free from clutter; you don't have to worry about things like curly braces and static typing that crop up in more complicated languages like Java.

With Python, you can just create code, run it, and get things done. Python is one of the languages found most commonly inside *The MagPi*, so learning it here will help you understand lots of the code used in projects.

IDE and IDLE

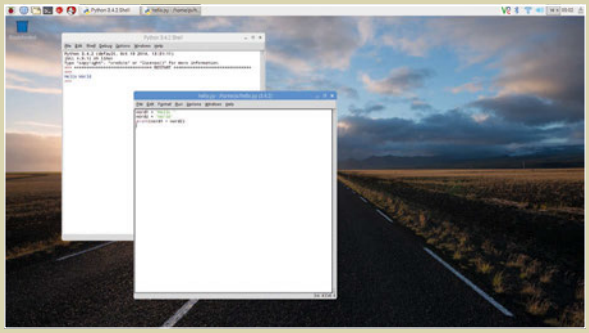
You don't have to write Python programs using a text editor like Leafpad and run them in the terminal. Instead, you can use a neat all-in-one solution, known as an 'IDE' (integrated development environment).

IDEs combine a text editor with program-running functionality. Often, they'll include fancy features like debugging and text completion.

Click **Menu > Programming > Python 3 (IDLE)**, and you'll get a new window called 'Python 3.4.2 Shell:'. This Shell works just like Python on the command line. Enter `print("Hello World")` to see the message.

You can also create programs in a built-in file editor. Choose **File > New File**. Enter this program in the window marked 'Untitled':

```
word1 = "Hello "
word2 = "World"
print(word1 + word2)
```



Above Python IDLE makes it easy to create programs and run them without having to use the command line

Don't forget to include the space after 'Hello'. Choose **File > Save As** and save it as **hello.py**. Now press **F5** on your keyboard to run the program. (Or choose **Run > Run Module**). It'll display 'Hello World' in the Shell.

The advantage of using Python IDLE is that you can inspect the program in the Shell. Enter `word1`, and you'll see 'Hello '. Enter `word2` and you'll see 'World'. This ability to inspect and use the variables in your program makes it a lot easier to experiment with programming and detect bugs (problems in your code).

Why Python?

There are a lot of programming languages out there, and they all offer something special. Python is a great option for beginners. Its syntax (the use of words and symbols) is easy to read. And it scales all the way up to industrial, medical, and scientific purposes, so it's ideal for beginners and experts alike.

Python in the terminal

You don't need to do anything to set up Python on your Raspberry Pi. Open a terminal in Raspbian and enter `python --version`. It will display 'Python 2.7.9'. Enter `python3 --version` and you'll see 'Python 3.4.2'.

We're going to use Python 3 in this feature (see 'Which Python?' boxout). You can open Python 3 in the terminal by just typing `python3`.

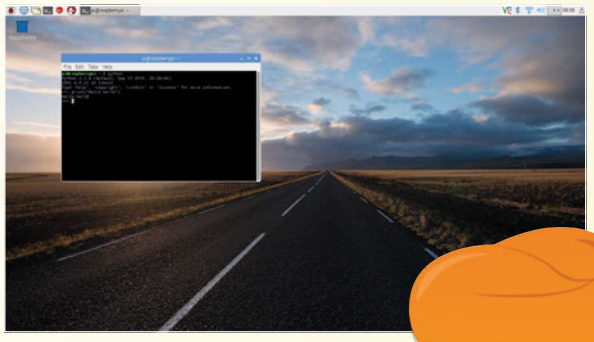
The '$' command-line prompt will be replaced with '>>>'. Here you can enter Python commands directly, just as you would terminal commands.

It's tradition to christen any new language by displaying 'Hello World'. Enter `print("Hello World")` and press **RETURN**. You'll see 'Hello World' outputted on the line below.

Using the Shell is known as Interactive Mode. You can interact directly with the code. It's handy for doing maths; enter `1920 * 1080` to get the answer: 2073600.

Mostly, you create Python programs using a regular text editor and save the files with a '.py' extension. Don't use a word processor like LibreOffice Writer, though – it'll add formatting and mess up the code.

Use a plain text editor like Leafpad (**Menu > Accessories > Text Editor**). Here you can enter your code, save it as a program, and then run the file in the terminal. Enter `python3 yourprogram.py` at the command line to run a program.



Left Python comes pre-installed in the Raspbian operating system and you can use it at the command line

Variables

Variables are all-purpose containers that you use to store data and objects

Python types

Python has five standard data types:

- Numbers
- String
- List
- Tuple
- Dictionary

Foo bar?

You'll come across 'foo' and 'bar' a lot when learning to code. These are dummy placeholders and don't mean anything. They could be zig and zag or bim and bam. Nobody's quite sure, but it might be related to the expression 'fubar' from the Vietnamese war.

If you've created a science project or experiment, you may have come across variables. In science, a variable is any factor that you can control, change or measure.

In computer programming, variables are used to store things in your program. They could be names, numbers, labels, and tags: all the stuff your program needs.

In Python, you write the name of a variable then a single equals sign and the word, number or object you want to put in it.

Enter this code directly into the Shell:

```
foo = 1
bar = 2
```

Remember: the variable name is on the left, and the thing it contains is on the right. Imagine you've got two plastic cups, and you've scrawled 'foo' on the first and 'bar' on the second. You put a number 1 in foo and a number 2 in bar.

If you ever want to get the number again, you just look in the cup. You do this in Python by just using the variable name:

```
foo
bar
```

You can also print out variables by passing them into a **print** function:

```
print(foo)
print(bar)
```

Variables can also be used to contain 'strings'. These are groups of letters (and other characters) that form words, phrases or other text.

Creating a string variable in Python is pretty much the same as creating an integer, except you surround the text with single (**' '**) or double (**" "**) quotes.

Using double quotes makes it easier to include apostrophes, such as **print("Don't worry. Be Happy")**. This line would break after 'Don' if you used single quotes – **print('Don't worry, be happy')** – so use double quotes for now.

Why variables count

Variables make it much easier to change parts of your code. Say you've got an excellent coding job at Nursery Rhymes Inc and you've written a classic:

```
print("Polly put the kettle on")
print("Polly put the kettle on")
print("Polly put the kettle on")
print("We'll all have tea")
```

The head of marketing comes in and says "our data shows that Polly isn't trending with the millennial demographic." You say "Huh!" and he barks "Change Polly to Dolly."

You now have to go through and change the variable in all three lines. What a downer! But what if you'd written thousands of lines of code and they all needed to change? You'd be there all week.

With variables, you define the variable once and then use it in your code. Then it's ready for changing

at any time:

```
name = "Polly"

print(name + " put the kettle on)
print(name + " put the kettle on)
print(name + " put the kettle on)
print("We'll all have tea")
```

This code prints out the same classic nursery rhyme. But if you want to change the name of our character, you only have to change it in one place:

```
name = "Dolly"
```

…and the poem will update on every line.

What's your type?

When you create a variable in Python, it's automatically assigned a type based on what it is. You can check this using the **type()** function. In the shell interface, enter:

```
foo = "Ten"
bar = 10
```

Now use the **type()** function to check the type of each variable:

```
type(foo)
type(bar)
```

It will say **<class 'str'>** for **foo**, and **<class 'int'>** for **bar**. This concept is important, because different types work together in a variety of ways, and they don't always play nicely together.

For example, if you add together two strings they are combined:

```
name = "Harry"
job = "Wizard"
print("Yer a " + job + " " + name)
```

This prints the message "Yer a Wizard Harry". The strings are concatenated (that's a fancy programming term for 'joined up'). Numbers, though, work completely differently. Let's try a bit of maths:

```
number1 = 6
number2 = 9

print(number1 + number2)
```

Instead of concatenating 6 and 9 together to give 69, Python performs a bit of maths, and you get the answer '15'.

Type casting

So what happens when you want to add a string and an integer together?

```
name = "Ben"
number = 10
print(name + number)
```

You'll get an error message: 'TypeError: Can't convert 'int' object to str implicitly'. This error is because Python can't add together a string and an integer, because they work differently. Ah, but not so fast! You can multiply strings and integers:

```
print(name * number)
```

It'll print 'Ben' ten times: you'll get 'BenBenBenBenBenBenBenBenBenBen'.

If you want to print out 'Ben10', you'll need to convert the integer to a string. You do this using a **str()** function and putting the integer inside the brackets. Here we do that, and store the result in a new variable called **number_as_string**:

```
number_as_string = str(number)
print(name + number_as_string)
```

This code will print out the name 'Ben10'. This concept is known as 'type casting': converting a variable from one type to another.

You can also cast strings into integers using the **int()** function. This is particularly useful when you use **input()** to get a number from the user; the input is stored as a string. Let's create a program that asks for a number and exponent and raises the number to the power of the exponent (using the '**' symbol):

```
number = input("Enter a number: ")
exponent = input("Enter an exponent: ")
result = int(number) ** int(exponent)
```

Our first two variables, **number** and **exponent**, are strings, while our third, **result**, is an integer. We could just print out the result:

```
print(result)
```

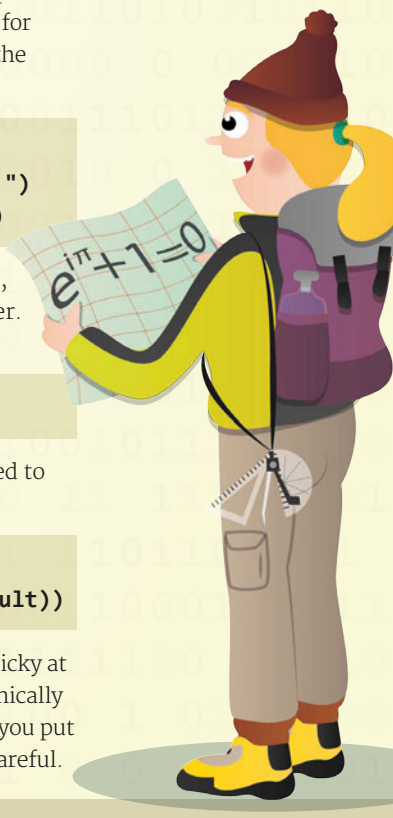
But if we wanted to include a message, we need to type cast **result** to a string:

```
print(number + " raised to the power
      " + exponent + " is " + str(result))
```

Variables, types, and type casting can be a bit tricky at first. Python is a lot easier to use because it dynamically changes the type of a variable to match the thing you put in it. However, it does mean you have to be a bit careful.

What to call a variable?

Variable names should be lower-case words separated by an underscore '_'. They can include numbers, but must start with a letter. You can call variables pretty much anything, but there's a small list of reserved keywords you should avoid (**magpi.cc/2h7MH1y**). It's a good idea to call them something that will be obvious when you use them in your program, like 'student_name' or 'person_age'.



Controlling flow with While & For

Get your program to do all the hard work with while and for loops

Comparison operators

These comparison operators are commonly used in conditions to determine if something is True or False:

==	equal
!=	not equal
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
<>	less than or greater than

Computers are great because they don't mind doing the same stuff over and over again. Their hard-working nature makes computers ideal for doing grunt work.

When looking at variables earlier, we printed out this nursery rhyme:

```
print("Polly put the kettle on")
print("Polly put the kettle on")
print("Polly put the kettle on")
print("We'll all have tea")
```

We didn't like the repetition of Polly, so we replaced it with a variable. But this code is foolish in another way: you have to write out the same **print** line three times.

We're going to use a loop to get rid of the repetition. The first loop we're going to look at is a 'while loop'. In Python 3 IDLE, create a new file and save it as **polly.py**; enter the code from the top of the next page.

We start with two variables:

```
name = "Polly"
counter = 0
```

Then we use the **while** statement followed by a condition: **counter < 3**.

On the next line down, you press the space bar four times to indent the code. Don't press the **TAB** key (see 'Tabs or spaces?' boxout).

```
while counter < 3:
    print(name + " put the kettle on")
    counter = counter + 1
```

The **<** symbol stands for 'less than'. It checks if the item on the left is less than the item on the right. In this case, it sees if the variable counter (which starts at 0) is less than 3. This condition is known as 'True'; if it wasn't, it'd be known as 'False'.

Finally, enter the last line of code:

```
print("We'll all have tea")
```

Save and run the program (press **F5**). It will print 'Polly put the kettle on' three times and then 'We'll all have tea'.

While, condition and indent

There are three things here: the while statement, the condition, and the indented text, organised like this:

```
while condition:
    indent
```

Imagine a three-way chat between all three items in our **polly.py** program:

Tabs or spaces?

There's a massive nerd debate about whether to use spaces or tabs when indenting code. There are valid arguments on both sides, which you can learn in this clip from HBO's comedy *Silicon Valley* (**magpi.cc/2gZde0M**). Use spaces for now. When you're a hardcore coder, you can make the argument for tabs.

While: "Hey Condition! What's your status?"
Condition: "True! The counter is 0. It's less than 3."
Indent: "OK, guys. I'll print out 'Polly put the kettle on' and increase the counter by 1. What's next?"

While: "Hey Condition. What's your status?"
Condition: "True! The counter is now 1."
Indent: "OK. I'm printing out another 'Polly put the kettle on' and increasing the counter by 1."

This goes on till the counter hits 3.

While: "Hey Condition. What's your status?"
Condition: "False! The counter is now 3, which isn't less than 3."
While: "OK guys. We're done!"

The program doesn't run the indented code, but moves to the single **print** at the end: 'We'll all have tea'.

For and lists

The next type of loop is known as 'for'. This is designed to work with lists.

Lists are a type of variable that contain multiple items (strings, numbers, or even other variables). Create a list by putting items inside square brackets:

```
banana_splits = ["Bingo", "Fleegle",
                 "Drooper", "Snorky"]
```

Now enter **banana_splits** in the Shell to view the list. It will display the four names inside the square brackets. You can access each item individually using the variable name and square brackets. Enter:

```
banana_splits[0]
```

...and you'll get 'Bingo'. Lists in Python are zero-indexed; that means the first item in the list is [0]. Here are each of the items. Type them into the Shell to get the names returned:

Polly.py

```
name = "Polly"
counter = 0

while counter < 3:
    print(name + " put the kettle on")
    counter = counter + 1

print("We'll all have tea")
```

```
banana_splits[0] # "Bingo"
banana_splits[1] # "Fleegle"
banana_splits[2] # "Drooper"
banana_splits[3] # "Snorky"
```

Zero-indexed lists can be confusing at first. Just remember that you're counting from 0. A for loop makes it easy to iterate over items in a list. Create this program and save it as **splits.py**:

```
banana_splits = ["Bingo", "Fleegle",
                 "Drooper", "Snorky"]

for banana_split in banana_splits:
    print(banana_split)
```

It doesn't matter what you use as the variable in a for loop, as long as you remember to use it in your indented code. You could put:

```
for dude in banana_splits:
    print(dude)
```

It's common to name the list as something plural (such as 'names', 'pages', and 'items') and use the singular version without the 's' for the 'in' variable: 'for name in names', 'for page in pages', and so on.

Infinite loops

You must be careful to change the counter in a while loop, or you'll get an infinite loop. If you delete the line **counter = counter + 1** from our while loop, it will run forever: it never goes above 0, so the indented code runs over and over again. This bug is known as an 'infinite loop' and is a bad thing to have in your programs.

Conditional Branching

Give your programs some brains with conditional branching

Logical operators

You can combine conditions together using logical operators.

and Both operands are true: (a and b) is True

or Any operator is true: (a or b) is True

not Checks if something is false: not (a and b) is True if both a and b are False.

Your programs have been slowly getting more powerful. We've learned to run instructions in procedural order, replaced parts of our program with variables, and looped over the code.

But another important part of programming is called 'conditional branching'. Branching is where a program decides whether to do something or not.

Of course, a program doesn't just decide whether or not to do things on a whim: we use the sturdy world of logic here.

The start of all this is the powerful 'if' statement. It looks similar to a loop, but runs just once. The if statement asks if a condition is True. If it is, then it runs the indented code:

```
if True:
    print("Hello World")
```

Run this program, and it'll display 'Hello World'. Now change the if statement to False:

```
if False:
    print("Hello World")
```

...and nothing will happen.

Of course, you can't just write True and False. Instead, you create a condition which evaluates to True or False; a common one is the equals sign (==). This checks whether both items on either side are the same. Create a new file and enter the code from **password1.py**. This code is a simple program that asks you to enter a password; if you enter the correct password, 'qwerty', it displays 'Welcome'.

Be careful not to confuse the equals logic operator == with the single equals sign =. While the double equals sign checks that both sides are the same, the single equals sign makes both sides the same. Getting == and = mixed up is a common mistake for rookie coders.

What else

After if, the next conditional branch control you need to learn is 'else'. This command is a companion to if and runs as an alternative version. When the if branch is True, it runs; when the if branch is False, the else branch runs.

```
if True:
   print("The first branch ran")
else:
   print("The second branch ran")
```

Run this program and you'll see 'The first branch ran'. But change True to False:

```
if False:
   print("The first branch ran")
else:
   print("The second branch ran")
```

...and you'll see 'The second branch ran'. Let's use this to expand our password program. Enter the code from **password2.py**.

Run the program again. If you get the password correct now, you'll get a welcome message. Otherwise, you'll get an 'incorrect password' message.

Elif

The third branching statement you need to know is 'elif'. This statement stands for 'else if', and sits between the if and else statements. Let's look at an elif statement. Enter this code:

```
if False:
    print("The first block of code ran")
elif True:
    print("The second block of code ran")
else:
    print("The third block of code ran")
```

Run this program and you'll find it skips the first if statement, but runs the elif statement. You'll get 'The second block of code ran'.

The else statement doesn't have a True or False condition; it runs so long as neither the if or elif statements are True. (Note that the else statement here, as always, is optional; you can just have if and elif.)

But what happens if you change both the if and elif conditions to True? Give it a try and see whether just if runs, or elif, or both. Experiment with removing the else statement and play around. It'll help you get the hang of the if, elif, and else statements.

FizzBuzz

We're going to show you a common program used in computer programming interviews. It's a classic called 'FizzBuzz', and it shows that you understand if, else, and elif statements.

First, you need to know about the modulo operator (**%**). This is used to get the remainder from a division and is similar to a divide operator. Take this function:

```
10 / 4 == 2.5
```

If we use a modulo instead, we get this:

```
10 % 4 == 2
```

Modulo turns out to be handy in lots of ways. You can use **% 2** to figure out if a number is odd or even:

```
10 % 2 == 0 # this is odd
11 % 2 == 1 # this is even
```

This program works out if a number is odd or even:

```
number = 10

if number % 2 == 0:
    print("The number is even")
else:
    print("The number is odd")
```

OK – let's move on to FizzBuzz.

Password.py

```
password = "qwerty"
attempt = input("Enter password: ")

if attempt == password:
    print("Welcome")
```

Password2.py

```
password = "qwerty"
attempt = input("Enter password: ")

if attempt == password:
    print("Welcome")
else:
    print("Incorrect password!")
```

Writing FizzBuzz

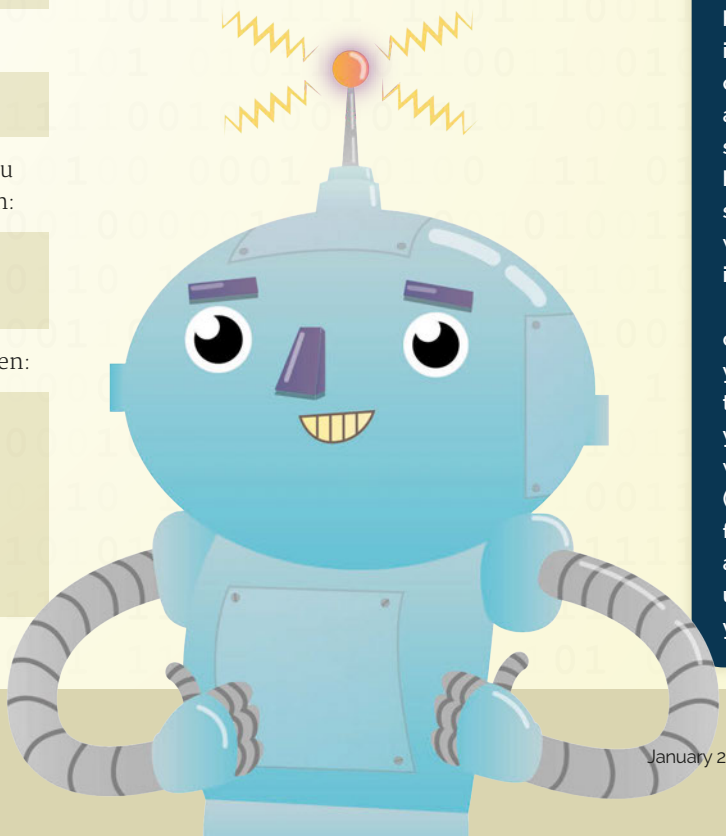
The brief for our FizzBuzz is to print the numbers from 1 to 100. If a number is divisible by three (such as 3, 6, and 9), then you print 'Fizz' instead of the number; if the number is divisible by five, you print 'Buzz' instead.

But if a number is divisible by both 3 and 5, such as the number 15, then you print 'FizzBuzz'.

We're also introducing a new element in FizzBuzz: the 'and' statement. This checks if two conditions are both True: that the number can be divided by both 3 and 5. It only returns True if both conditions are True.

There are three main logical operators: and, or, and not. The first two are relatively straightforward, but the 'not' operator can be more confusing at first. Don't worry about it too much; you'll get the hang of it with practice.

Enter the **fizzbuzz.py** code from page 25 to practise using if, else, and elif elements and logical operators.



Comments

A mark of a good programmer is to use comments in your programs. Comments are used to explain bits of your program to humans. They are completely ignored by the computer.

In Python, you start a comment line with a hash symbol (#). It can be on a line on it own, or it can come right after a line of code. As soon as Python hits the #, it'll stop translating whatever follows into machine code.

Comments help other users to read your program, but they will also help you understand what you're doing (long after you've forgotten). It's a good habit to use comments in your programs.

Creating Functions

Create the building blocks of code and make more robust programs

You've come a long way since your first 'Hello World'. Your programs now check for conditions and loop over themselves.

You're now writing programs that are known as 'Turing complete', named after Alan Turing, the father of computer science and artificial intelligence, who hacked the German Enigma code in WWII.

Now we're going to take things a little further. We're going to introduce you to a form of modularity called functions.

Functions are blocks of code that you write once and can repeat anywhere. It's a little like being able to write a block of text once, and then paste it whenever you need it.

Spotting a function

Python is packed with built-in functions, and you've already been using them in your programs. Commands like **`print()`**, **`len()`**, and **`type()`** are all functions. They're easy to spot: a small command starting with a lower-case letter and followed by a pair of parentheses '()'.

Python documentation

You can browse or download a copy of the Python documentation directly from the Python website at **python.org/doc**. Python has a whole bunch of built-in functions. You can view a list of all the built-in functions on the Python documentation website (**magpi.cc/2gPsGK3**).

Using functions

Let's take a look at a function called **`abs()`**. It stands for 'absolute', and returns the absolute value of any number you pass into it (the bit you pass in is called the 'argument').

An absolute number is the positive of any number, so if you write **`abs(-2)`** you get 2 back. Try this in the Shell:

```
abs(2) # returns 2
abs(-2) # returns 2
```

You can store the returned result as a variable:

```
positive_number = abs(-10)
```

We find it easier to read a function backwards, from right to left. The value is passed into the parentheses, then the function cranks it and returns a new value. This is passed left and stored into the variable.

Defining a function

The great thing about Python is that you don't just use the built-in functions: you get to make your own. These are called 'user-defined functions'.

You create a function using the **`def`** keyword, followed by the function name and parentheses. Inside the parentheses, you list the parameters. These are the same as the arguments, only inside the definition they are called 'parameters'.

```
def function(parameter):
    return parameter
```

Our function here doesn't do anything; it simply accepts a parameter and returns it.

At the end of the function definition is a colon (**:**). The function code is indented by four spaces, just like a loop or if/else branch.

The code inside the indentation runs when you call the function. Functions typically include a **return** statement which passes back an expression.

Working functions

We're going to create a function that prints the lyrics to Happy Birthday.

Type out the **happy_birthday.py** code from the listing, then run it. In the Shell, enter:

```
happy_birthday("Lucy")
```

This function call uses the string 'Lucy' as the argument. This string is passed into the function as the parameter and is then available for use in the indented code inside the function.

Return statements

Many functions don't just run a block of code; they also return something to the function call.

We saw this in **abs()**, which returned the absolute value of a number. This can be stored in a variable.

In fact, we're going to recreate the **abs()** function, so you can see how it's working behind the scenes.

In maths, you invert a positive/negative value by multiplying a negative number by -1, like this:

```
10 * -1 = -10
-10 * -1 = 10
```

We need to create a function that takes a number as a parameter and checks if it's negative. If so, it multiplies it by -1; if it's positive, it simply returns the number. We're going to call our function **absolute()**.

Enter the code in **absolute.py**. When the function hits either of the **return** statements, it returns the value of the number (either on its own or multiplied by -1). It then exits the function.

Run the **absolute.py** code and enter the following in the Shell:

```
absolute(10)
absolute(-10)
```

Our last program listing is a classic known as 'FizzBuzz'; as mentioned on page 23, it will help you to understand if, else, and elif.

You also need to know the modulo operator (**%**) for FizzBuzz. This operator returns the remainder from a division. If you don't know how modulo works, watch this video (**magpi.cc/2h5XNRO**).

Now work through the code in **fizzbuzz.py**.

Happy_birthday.py

```
def happy_birthday(name):
    count = 0
    while count < 4:
        if count != 2:
            print("Happy birthday to you")
        else:
            print("Happy birthday dear " + name)
        count += 1
```

Absolute.py

```
def absolute(number):
    if number < 0:
        return number * -1
    else:
        return number
```

Fizzbuzz.py

```
count = 0
end = 100

while count < end:
    if count % 5 == 0 and count % 3 == 0:
        print("FizzBuzz")
    elif count % 3 == 0:
        print("Fizz")
    elif count % 5 == 0:
        print("Buzz")
    else:
        print(count)

    count += 1
```

Going further

Here are some resources you will find useful.

GPIO Zero Essentials – magpi.cc/2bA3ZP7
This Essentials guide book explains how the GPIO Zero Python module provides access to a bunch of features. These are used to hook up electronics to your Raspberry Pi via the GPIO pins.

FutureLearn – magpi.cc/2h5Sthf
The Raspberry Pi Foundation has two new online training courses: Teaching Physical Computing with Raspberry Pi and Python, and Teaching Programming in Primary Schools.

Learning Python – magpi.cc/2h2opWC
This tutorial provided by The Raspberry Pi Foundation has files you can download. You download the file, called **intro.py**, using this command in a Terminal: `wget http://goo.gl/0ZDOdX -O intro.py --no-check-certificate`. Open the **intro.py** file in IDLE; all the instructions are in the file.

Importing Code

Stand on the shoulders of giants by importing other programmers' code

Pygame

If you want to learn more about Pygame, check out *Make Games With Python*, our free Essentials Guide to the Pygame module. **magpi.cc/2h2m0vh**

This being the modern world, you're not supposed to do all the work on your own. Instead, you will often stand on the shoulders of other programmers who have done the groundwork for you.

Your programs can import code created by other people using the **import** statement. This enables you to import modules and use their functions – only they're now known as 'methods'.

You import the module at the command line, and then access the functions using dot notation. This is where you list the module, followed by a dot (**.**), then the method.

A common module to use is **math**. This allows you to access lots of maths methods. Open a Python Shell and enter:

```
import math
```

You now have access to all the methods in **math**. You won't notice any difference, but if you type:

```
type(math)
```

...it will say '<class 'module'>'. Let's try out dot notation now. Type **math** followed by a dot and the name of the method (function) you want to use:

```
math.sqrt(16)
```

This gives the square root of 16, which is 4.

Some methods have more than one argument. The **math.pow()** method raises a number to an exponent:

```
math.pow(64,3)
```

This returns 262144.0.

You can also access constant values from a module, which are fixed variables contained in the module. These are like functions/methods, but without the parentheses.

```
math.pi
```

This returns pi to 15 decimal spaces: 3.141592653589793.

```
math.e
```

This returns Euler's number to 15 decimal spaces: 2.718281828459045.

It's also possible to import methods and constants from modules using **from**. This enables you to use them inside your programs without dot notation (like regular functions). For example:

```
from math import pi
from math import e
from math import pow
```

Now, whenever you type **pi** or **e**, you'll get pi and Euler's number. You can also use **pow()** just like a regular function. You can change the name of the function as you import it with **as**:

```
from math import pi as p
```

Now when you enter **p** you'll get pi to 15 decimal spaces. Don't go crazy renaming functions with **as**, but it's common to see some methods and constants imported as single letters.

By creating your own functions, and importing those created by other people in modules, you can vastly improve the capabilities of your programs.

We're going to take everything we've learnt and use it to create a game of Pong; this is one of the world's first videogames.

Write out the code carefully in **pong.py**. Here you'll find variables, functions, loops, and conditional branching: all the stuff we've talked about. Hopefully, you'll now be able to decipher most of this code.

If you're interested in taking Pong further, this program is similar to a version of a Pygame program by Trever Appleton (**magpi.cc/2hgkOUX**). His version has a scorecard and more advanced code. We've kept ours simple so it's easier to start with.

Hopefully this isn't the end of your Python, or programming, journey. There are lots of places you can learn programming from. And we'll have more programming resources for you in every issue of *The MagPi*.

Pong.py

```
import pygame, sys
from pygame.locals import *

# Set up game variables
window_width = 400
window_height = 300
line_thickness = 10
paddle_size = 50 # try making this smaller for a harder game
paddle_offset = 20

# Set up colour variables
black = (0  ,0  ,0  ) # variables inside brackets are 'tuples'
white = (255,255,255) # tuples are like lists but the values don't change

# Ball variables (x, y Cartesian coordinates)
# Start position middle of horizontal and vertical arena
ballX = window_width/2 - line_thickness/2
ballY = window_height/2 - line_thickness/2

# Variables to track ball direction
ballDirX = -1 ## -1 = left 1 = right
ballDirY = -1 ## -1 = up 1 = down

# Starting position in middle of game arena
playerOnePosition = (window_height - paddle_size) /2
playerTwoPosition = (window_height - paddle_size) /2

# Create rectangles for ball and paddles
paddle1 = pygame.Rect(paddle_offset,playerOnePosition, line_thickness,paddle_size)
paddle2 = pygame.Rect(window_width - paddle_offset - line_thickness, playerTwoPosition, line_thickness,paddle_size)
ball = pygame.Rect(ballX, ballY, line_thickness, line_thickness)

# Function to draw the arena
def drawArena():
    screen.fill((0,0,0))
    # Draw outline of arena
    pygame.draw.rect(screen, white, ((0,0),(window_width,window_height)), line_thickness*2)
    # Draw centre line
    pygame.draw.line(screen, white, ((int(window_width/2)),0),((int(window_width/2)),window_height), (int(line_thickness/4)))

# Function to draw the paddles
def drawPaddle(paddle):
    # Stop the paddle moving too low
    if paddle.bottom > window_height - line_thickness:
        paddle.bottom = window_height- line_thickness
    # Stop the paddle moving too high
    elif paddle.top < line_thickness:
        paddle.top = line_thickness
    # Draws paddle
    pygame.draw.rect(screen, white, paddle)

# Function to draw the ball
def drawBall(ball):
    pygame.draw.rect(screen, white, ball)

# Function to move the ball
def moveBall(ball, ballDirX, ballDirY):
    ball.x += ballDirX
    ball.y += ballDirY
    return ball # returns new position

# Function checks for collision with wall and changes ball direction
def checkEdgeCollision(ball, ballDirX, ballDirY):
    if ball.top == (line_thickness) or ball.bottom == (window_height - line_thickness):
        ballDirY = ballDirY * -1
    if ball.left == (line_thickness) or ball.right == (window_width - line_thickness):
        ballDirX = ballDirX * -1
    return ballDirX, ballDirY # return new direction

# Function checks if ball has hit paddle
def checkHitBall(ball, paddle1, paddle2, ballDirX):
    if ballDirX == -1 and paddle1.right == ball.left and paddle1.top < ball.top and paddle1.bottom > ball.bottom:
        return -1 # return new direction (right)
    elif ballDirX == 1 and paddle2.left == ball.right and paddle2.top < ball.top and paddle2.bottom > ball.bottom:
        return -1 # return new direction (right)
    else:
        return 1 # return new direction (left)

# Function for AI of computer player
def artificialIntelligence(ball, ballDirX, paddle2):
    # Ball is moving away from paddle, move bat to centre
    if ballDirX == -1:
        if paddle2.centery < (window_height/2):
            paddle2.y += 1
        elif paddle2.centery > (window_height/2):
            paddle2.y -= 1
    # Ball moving towards bat, track its movement
    elif ballDirX == 1:
        if paddle2.centery < ball.centery:
            paddle2.y += 1
        else:
            paddle2.y -=1
    return paddle2

# Initialise the window
screen = pygame.display.set_mode((window_width,window_height))
pygame.display.set_caption('Pong') # Displays in the window

# Draw the arena and paddles
drawArena()
drawPaddle(paddle1)
drawPaddle(paddle2)
drawBall(ball)

# Make cursor invisible
pygame.mouse.set_visible(0)

# Main game runs in this loop
while True: # infinite loop. Press Ctrl-C to quit game
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
        # Mouse movement
        elif event.type == MOUSEMOTION:
            mousex, mousey = event.pos
            paddle1.y = mousey

    drawArena()
    drawPaddle(paddle1)
    drawPaddle(paddle2)
    drawBall(ball)

    ball = moveBall(ball, ballDirX, ballDirY)
    ballDirX, ballDirY = checkEdgeCollision(ball, ballDirX, ballDirY)
    ballDirX = ballDirX * checkHitBall(ball, paddle1, paddle2, ballDirX)
    paddle2 = artificialIntelligence (ball, ballDirX, paddle2)
    pygame.display.update()

```