Fizzbuzz Challenge:
An introduction to programming in Python

Objective: To learn basic programming concepts and Python syntax through a classic programming challenge.

1 Introduction

The Fizzbuzz game is often used in UK schools to teach children about division. The rules are very simple: starting from 1, the children take turns saying the next number. However, if the number is divisible by 3, they must instead say "Fizz". If the number is divisibly by 5, they must say "Buzz". And if the number is divisible by both 3 and 5, the child must say "Fizzbuzz".

A version of this game has become a classic programming exercise, even used for some programming job interviews. It involves writing a program that "plays" the first 100 turns of the Fizzbuzz game with itself. In other words, the program must display a list of the first 100 numbers, replacing multiples of 3 with Fizz, multiples of 5 with Buzz, and multiples of 3 and 5 with Fizzbuzz.

Using the free, widely-used programming language Python, we're going to learn the concepts needed to solve this challenge. To get the most out of this exercise, it is important to have access to a Python interpreter (a program that can read and understand Python code) and try it for yourself. The easiest way to do this is to connect to the website `https://repl.it/languages/python3`. On the right-hand side of the screen is an interactive Python interpreter: you can type a line of code and press Return to see what happens (try writing 2+2 and hitting Return, for instance). On the left of the screen is a text editor window. In here you can type a longer program, and click the ▶ to run the code.

Once all of the tools concepts are in place, you will be given warning before solutions are given, so you can try to solve it yourself. If you fly through the task, there will be alternative solutions implementing more concepts further down. You are fully encouraged to experiment, and not follow these instructions word-for-word. Finally got something working? Now try and break it! This is how we learn programming.

2 The tools we need to solve the challenge

2.1 Arithmetic

All programming languages offer the programmer basic arithmetic operations such as addition, multiplication, and so on. On the right-hand side of the screen, in the interpreter window, try some of the following operations to verify that they work. You can use integers (whole numbers) or numbers with a decimal point for the following operations:

Operation	Python symbol	Example
Addition	`+`	`2 + 2`
Subtraction	`-`	`7.5 - 2`
Multiplication	`*`	`3 * 5`
Division	`/`	`7 / 3`
Exponent	`**`	`3**2`

In this table, `3**2` stands for 3^2 in standard mathematical notation. There are two more arithmetic operations that only work on integers (which in Python means numbers with no decimal point: 5.0 does not count as an integer!). Recall that when you first learned division in school, you would have learned that 7 divided by 3 is 2, with a remainder of 1. In this sentence, 2 is called the quotient, and 1 is the remainder. Python has operations for getting the quotient and remainder of a division:

Integer Operation	Python symbol	Example
Quotient	`//`	`7 // 3`
Remainder	`%`	`7 % 3`

If you try to use these operations when one of the numbers has a decimal point, you will get an error. To do more complicated calculations, you can use brackets `(` and `)` to specify the order to evaluate the operations, just like in normal mathematical expressions.

Since our Fizzbuzz challenge is all about whether integers are divisible by certain numbers, consider it a hint that these last two operations might be very useful!

2.2 Variables, and printing words

An important action in most programming languages is giving names to pieces of information we would like to refer to later. This is called *assigning a variable*. Let's say we'd like the `a` to refer to the number `3`. The code is simply as follows:

```
a = 3
```

A program which contains this line can now use the letter `a` to mean the number `3` at any time. You can use almost any letter or word as a variable, including Chinese characters, unless it is one of Python's special keywords that already has a meaning.

Another common operation is to display some information on the screen. To do this, we use the `print()` function; the thing we want to display goes inside the brackets. So, on the left-hand side of the screen, in the editor window, we can try to write our first multi-step program. Type the following code, and then hit the ▶ symbol:

```
luckiest_number = 8
print("The luckiest number is:")
print(luckiest_number)
```

Look closely at the second line. The text we wish to display is wrapped in quotation marks " to signify that this is not programming code, but text meant to be read by humans. This is called a Python string. Again, Chinese characters are quite acceptable in a Python string.

Any piece of information can be stored in a variable, so for instance:

```
my_name = "Sam"
print("My name is", my_name)
```

will write `My name is Sam` on the screen. You can also re-assign a variable at any time:

```
my_name = "Sam"
my_name = "Sam Cassidy"
print(my_name)
```

to display `Sam Cassidy`. Variables can also reference other variables, even themselves!

```
a = 3
a = a + 1
```

means that `a` now equals `4`. The `=` sign does not mean the same thing in Python as it does in mathematics!

2.3 True or False?

When programming, we don't always want our program to just follow a list of instructions line-by-line. Sometimes, we want our program to make a decision: do A, or do B. To do this, we check whether a certain fact is true or false. If it is true, we do A. If it is false, we do B.

In Fizzbuzz, the children (or our program) has to decide whether to say the number, or say Fizz or Buzz. Here we'll learn how to make that decision.

I recommand that you try all the examples by writing them in the editor window on the left of the screen and pressing ▶. Make sure to change things and break it so you can see how it works! If you get an error, try to figure out why, or ask a friend.

2.3.1 if

What we need is to use an `if` block. If we start a line with `if <something>:` and then *indent* the next few lines, then those indented lines will run only if the `<something>` is true. For example:

```
if 5 > 3:
    print("It worked!")
print("This code will always run")
```

The indented line will only run if $5 > 3$ is true, which it is! So both lines should run. However, if we swap the $>$ around, we have:

```
if 5 < 3:
    print("It worked!")
print("This code will always run")
```

This time, the line `print("It worked!")` won't do anything. The program just ignores it, because the line in the header is not true. Of course, there is no mystery here as to whether that line will run. Let's do something more interesting. To get an input from the user, you can use the `input()` function. Inside the brackets, you can put a message to the user to tell them what to write. Let's write a program that can tell the user if a number they select is even, or odd. When the user enters something, the program will first see it as a string, so we need to tell it to think of it as an integer.

```
# NOTE: Python ignores anything that comes after a # sign.
# You can use this to write notes and comments in your code.

number = input("Please enter a number: ")
number = int(number) # converts it to an integer
if number % 2 == 0:
    print("Your number is even!")
```

Remember what the `%` means: the remainder after division. So this program detects whether the number is even by asking if it gives remainder 0 after division by 2. Note that to check for equality, we have to use `==` instead of `=`, because `=` is used to assign variables.

2.3.2 else

So what about the other case? We want the program to do something different when the number is odd, right? In that case, we use the word `else` in the same way we use the word `if`:

```
number = input("Please enter a number: ")
number = int(number) # converts it to an integer
if number % 2 == 0:
    print("Your number is even!")
else:
    print("Your number is odd!")
```

The indentation (white space at the beginning of the line) is very important; it's how Python code is organized into sections called blocks. Now our program can make a simple yes/no decision. But what if there are more choices?

2.3.3 elif

To add more options, you can use the `elif` keyword. It's short for "else-if", and works just like "if", but it is only checked if the previous check came out False. Let's write a similar program, but that this time tells the user if their number is positive, negative, or zero.

```
number = input("Please enter a number: ")

# the next line converts it to a decimal number
# (this time we allow decimals!)
number = float(number)

if number > 0:
    print("Your number is positive!")
elif number < 0:
```

```
    print("Your number is negative!")
else:
    print("Your number must be zero!")
```

We can use `elif` as many times as we like, to create as many branches in our program as we like.

2.3.4 Multiple conditions

The keywords `and` and `or` can be used to check more than one condition before making a decision. If we use `and`, then *both* conditions must be true to pass the test. If we use `or`, only one condition has to be true. For example:

```
number = 5
if number < 7 and number > 0:
    print("This will print!")
if number < 7 and number < 3:
    print("This won't print!")
if number < 7 or number < 3:
    print("This will, however!")
```

Experiment with this! See what you can come up with!

2.4 Repeating an action

Now, it would be disappointing if, when we come to solve the Fizzbuzz challenge, we had to write a whole new section of code for every number between 1 and 100. Thankfully, it's easy to write code that repeats itself, with a little change each time.

A for-loop repeats a certain action for each item in a sequence. So for example, Python considers a string to be a sequence of letters. So we could use a for-loop to do an action on each letter. The syntax is quite similar to the `if` syntax we met last time:

```
for letter in "XJTLU":
    print(letter)
```

will give the output

```
X
J
T
L
U
```

It is important to understand that here, "`letter`" is just a temporary variable, which we can set to anything we like. Each time the loop repeats itself, the variable `letter` is set to the next letter. So this code is exactly the same:

```
for banana in "XJTLU":
    print(banana)
```

Now, in our Fizzbuzz game, we want to work with a sequence of numbers: those 1 to 100. To do this, Python gives us a function called `range()`. Inside the brackets, we specify two numbers, and `range` gives us a sequence of numbers between them. For example:

```
for x in range(1, 6):
    print("Hello")
    print(x)
print("Done")
```

will give us

```
Hello
1
Hello
2
Hello
3
Hello
```

```
4
Hello
5
Done
```

Neat! As always, try out the examples in the editor window and press ▶ to see how it works, and then experiment with making your own loops! You can use deeper levels of indentation to include if-statements inside loops, or even loops within loops! For example:

```
for x in range(1, 6):
    for y in range(1, x):
        print(y)
```

What does this one do?

3 Solving the task

Believe it or not, you now have all the ingredients to solve the Fizzbuzz task. The next step involves using your brain to solve the problem. Remember, you're trying to produce a list like

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
```

⋮

by combining the ideas we've learned so far in the right way. This will require you design the program, and figure out how to code it correctly in Python! Best of luck!

The solution will be printed following an empty page, if you get very stuck.

4 The solution

One correct solution to the problem is as follows:

```
for x in range(1, 101):
    if x % 3 == 0 and x % 5 == 0:
        print("Fizzbuzz")
    elif x % 3 == 0:
        print("Fizz")
    elif x % 5 == 0:
        print("Buzz")
    else:
        print(x)
```

The trickiest part, in my opinion, is figuring out that you have to check for divisibility by both 3 and 5 (Fizzbuzz) before checking for divisibility by 3 (Fizz) or 5 (Buzz). Can you think why?

5 Extra: Going further

Our solution to the task is only one of many possible methods. Here, we'll survey some other Python programming concepts that can be used to solve the task, to give you a flavour of other parts of Python. As always, experiment with these ideas yourself, but don't worry if things seem a bit confusing: we are getting a little ahead of ourselves. The first concept we'll look at is "encapsulating" the task into a function.

5.1 Functions

An important concept in programming is that once a problem has been solved, that solution can be reused again and again without having to worry about reprogramming it each time. In fact, that's the basic idea of software: someone else has written the program, and you can use it whenever you need to, without programming it yourself or worrying how it works.

A *function* is a section of code that can be reused in your program (or another program), allowing a complex task to be executed with just a single word. For example, I can write a very simple function that just says hello to me:

```
def hello_Sam():
    print("Hello, Sam!")
```

The word "def" is short for define, and it means that we are defining a new function. As usual, we use the indentation to tell the Python interpreter software when the function begins and ends. Simply defining the function does nothing in the program however. We're just telling the Python interpreter what the word `hello_Sam` means. From now on, however, anywhere in my code, I can write `hello_Sam()` to execute the code in the function. This is called *calling* the function.

Not very useful, I admit. I could write another function that says hello to someone else

```
def hello_someoneelse():
    print("Hello, Zhang Li!")
```

Again, not very useful, but it means I can write `hello_someoneelse()` anywhere in my code to print that greeting. Here's the punchline, though. Our two functions are performing a very similar task, only with a slight change to the phrase. What we should really do is write a function that can greet *anybody*, by allowing the user to specify an input. The inputs to a function are called arguments. Read the example code, and try this out for yourself.

```
def say_hello(name):
    print("Hello, " + name + "!")
```

In this example, "`name`" is acting as a dummy input to the function. It has no meaning on its own, but now, if I call the function with `say_hello("Sam")`, I get "Hello, Sam!", and if I call the function with `say_hello("Zhang Li")`, I get "Hello, Zhang Li!". The same code is reused, but with a different input.

This means the Fizzbuzz code above can be re-structured to use a function, if I should please. The input to the function should be the number. But functions can also have an output, called a *return value*. Simply printing a result is not really considered a return value, because we cannot actually do anything

with what is printed. The user can read it, but the rest of the program cannot use it. Instead, we use the keyword `return` to tell the function what to output. Then we can decide, independently of the function: do we want to print the output? Do we want to store it for later? Do we want to immediately send that output as the input of another function? Let's rewrite Fizzbuzz using this idea:

```
# This part defines the function
def fizzbuzz(x):
    if x % 3 == 0 and x % 5 == 0:
        return "Fizzbuzz"
    elif x % 3 == 0:
        return "Fizz"
    elif x % 5 == 0:
        return "Buzz"
    else:
        return str(x)

# Here is the code that actually runs
for k in range(1, 101):
            print(fizzbuzz(k))
```

(Note: here I chose to return `str(x)` in the `else` clause, which will turn the number `x` into the string `"x"`. This is because it is a good habit that functions should always return the same kind of data. Without this `str()` bit, the function would sometimes return a string and sometimes a number.)

Now we can Fizzbuzz any number, at any time. For instance, `fizzbuzz(1232442)` gives the output `"Fizz"`. This is, of course, a frivolous example, but functions that can take an input and give an output, without the programmer having to worry about what goes on in between, is the heart of programming. We have seen, for example, the built-in `range()` function, which takes a pair of numbers as an input and gives the sequence of numbers between them as an output. Do we know how it works? No. Do we need to? No!

Programmers often share the functions they have made to solve a particular task in collections called libraries. Python has a huge selection of libraries for solving all kinds of problems, from building websites, to analysing DNA sequences.

5.2 Storing data

So far, we have been simplying putting the results of the program on the screen. Another options is to store the data in a *structure* – a way of organizing information in a program. There are many kinds of structure, but the simplest, and most appropriate for storing our fizzbuzz results is called a *list*. In Python, a list is a sequence of data that can be extended, shrunk, sorted, and otherwise modified in many ways. To create a list of items, we simply place them in square brackets separated by commas:

```
hipy_beekeepers = ["Rob", "Laurie", "Samantha", "Ben",
"Marco", "Meera", "Cate", "Fiona",
"Chris", "Dan", "Zeena", "Eiman",
"Wille", "Sam C", "Sam B"]
```

Try creating a list in Python. The entries can be anything, maybes places you've visited, or foods you like, numbers (you don't need to use ""s if they are numbers – they're only for text strings). Now you can access the entries of the list by using the name of the list, followed by `[x]`, where `x` is the number of the element of the list (starting at 0). For instance, in my list, `hipy_beekeepers[0]` refers to Rob, while `hipy_beekeepers[2]` refers to Samantha. I can get sections of the list by specifying a range: `hipy_beekeepers[0:3]` refers to `["Rob", "Laurie", "Samantha"]`. The list entries can be modified. For instance, `hipy_beekeepers[13] = "Sam Cassidy"` will change the 14th entry to Sam Cassidy. A new entry can be added with `hipy_beekeepers.append("Felix")`. Now when I type:

```
print(hipy_beekeepers)
```

I see

```
["Rob", "Laurie", "Samantha", "Ben",
"Marco", "Meera", "Cate", "Fiona",
"Chris", "Dan", "Zeena", "Eiman",
"Wille", "Sam C", "Sam B", "Felix"]
```

How about alphabetical order? Well,

```
hipy_beekeepers.sort()
print(hipy_beekeepers)
```

will give me

```
["Ben", "Cate", "Chris", "Dan",
"Eiman","Felix", "Fiona", "Laurie",
"Marco", "Meera", "Rob", "Sam B",
"Sam C", "Samantha", "Wille", "Zeena"]
```

So we can already see that lists are powerful and flexible ways of storing a large amount of data in a single variable. There are two easy ways we could put all of our Fizzbuzz results into a list. The first is easier to understand, but computationally quite slow, because modifying the length of a list is quite a slow operation:

```
def fizzbuzz(x):
    if x % 3 == 0 and x % 5 == 0:
        return "Fizzbuzz"
    elif x % 3 == 0:
        return "Fizz"
    elif x % 5 == 0:
        return "Buzz"
    else:
        return str(x)

fizz_list = [] # an empty list!
for k in range(1, 101):
    entry = fizzbuzz(k)
    fizz_list.append(entry) # add the results of the function to the list

print(fizz_list)
```

Try writing this code for yourself and see how it works. Now, there is a more efficient and elegant way to produce this list. This is a tool called a "list comprehension". We essentially bring the for-loop *inside* the list creation. Don't worry if you do not quite understand the following code: this is just a taste of what Python can do.

```
def fizzbuzz(x):
    if x % 3 == 0 and x % 5 == 0:
        return "Fizzbuzz"
    elif x % 3 == 0:
        return "Fizz"
    elif x % 5 == 0:
        return "Buzz"
    else:
        return str(x)


fizz_list = [fizzbuzz(k) for k in range(1, 101)]

print(fizz_list)
```

With list comprehensions, complex data structures can be created quickly with a single line of code!

6 Closing remarks

However far you got into the challenge, I hope you have had fun and seen that a little programming skill can go a long way, even if you're not a professional programmer. If you want to do calculations in finance, science, or engineering, or maybe automate tasks on your computer to make you more efficient at work, programming can help you do that. Python is a great first language. If you're interested in learning more, why not install the Python interpreter on your computer and experiment further? HiPy Liverpool will be sharing more of the resources we use with XJTLU, where you can learn more "real" programming to solve real problems.