Part 2: Functions and Control Statements

A distinguishing property of *programming* languages is that the programmer can create their own *functions*. Creating a *function* is like teaching the computer a new trick. Typically a function will receive some data as *input*, will perform an *algorithm* involving the input data, and will *output* data when the algorithm terminates.

In this part, we explore Python functions. We also explore control statements, which allow a program to behave in different ways for different inputs. We also introduce the *while loop*, a loop whose repetition can be more carefully controlled than a for loop. As an application of these techniques, we implement the Euclidean algorithm as a Python function in a few ways, to effectively find the GCD of integers.

At the end, you will be prepared to explore the Collatz conjecture.

Getting started with Python functions

A *function* in Python is a construction which takes input data, performs some actions, and outputs data. It is best to start with a few examples and break down the code. Here is a function `square`. Run the code as usual by pressing *shift-Enter* when the code block is selected.

```
def square(x):
    answer = x * x
    return answer
```

To see "under the hood" what Python is doing, we import the Python disassembler.

```
from dis import dis
```

```
dis(square)
```

```
  2           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                0 (x)
              4 BINARY_MULTIPLY
              6 STORE_FAST               1 (answer)

  3           8 LOAD_FAST                1 (answer)
             10 RETURN_VALUE
```

When you run the code block, you probably didn't see anything happen. But you have effectively taught your computer a new trick, increasing the vocabulary of commands it understands through the Python interpreter.

More specifically, Python has turned your neat "square function" into a series of very quickly runnable commands ("LOAD_FAST" and "BINARY_MULTIPLY"). That way, every time you want to use your square function, Python will just go through the same series of quick operations.

You don't really need to know the "disassembled" square function above... you can use the `square` command as you wish.

```
square(12)
```

```
144
```

```
square(1.5)
```

```
2.25
```

Let's break down the syntax of the *function declaration*, line by line.

```
def square(x):
    answer = x * x
    return answer
```

The first line begins with the Python reserved word `def`. (So don't use `def` as a variable name!). The word `def` stands for "define" and it defines a function called `square`. After the function name `square` comes parentheses `(x)` containing the **argument** `x`. The *arguments* or *parameters* of a function refer to the input data. Even if your function has no arguments, you need parentheses. The argument `x` is used to name whatever number is input into the `square` function.

At the end of the function declaration line is a colon `:` and the following two lines are indented. As in the case of for loops, the colon and indentation are signals of *scope*. Everything on the indented lines is considered within the *scope of the function* and is carried out when the function is used later.

The second line `answer = x * x` is the beginning of the scope of the function. It declares a variable `answer` and sets the value to be `x * x`. So if the argument `x` is 12, then `answer` will be set to 144. The variable `answer`, being declared within the scope of the function, will not be accessible outside the scope of the function. It is called a **local variable**.

The last line `return answer` contains the Python reserved word `return`, which terminates the function and outputs the value of the variable `answer`. So when you apply the function with the command `square(1.5)`, the number `1.5` is `passed` as the argument `x`, and `answer` is `2.25`, and that number `2.25` becomes the output.

A function does not have to return a value. Some functions might just provide some information. Here is a function which displays the result of division with remainder as a sentence with addition and multiplication.

```
def display_divmod(a,b):
    quotient = a // b # Integer division
    remainder = a % b #
    print("{} = {} ({}) + {}".format(a,quotient,b,remainder))
```

If you want a quick look under the hood, here is the function disassembled. Again, you don't need to know what everything means. The four-line code above is translated into 20 very fast steps below, numbered 0,2,4, up to 38.

```
dis(display_divmod)
```

```
  2           0 LOAD_FAST                0 (a)
              2 LOAD_FAST                1 (b)
              4 BINARY_FLOOR_DIVIDE
              6 STORE_FAST               2 (quotient)

  3           8 LOAD_FAST                0 (a)
             10 LOAD_FAST                1 (b)
             12 BINARY_MODULO
             14 STORE_FAST               3 (remainder)

  4          16 LOAD_GLOBAL              0 (print)
             18 LOAD_CONST               1 ('{} = {} ({}) + {}')
             20 LOAD_ATTR                1 (format)
             22 LOAD_FAST                0 (a)
             24 LOAD_FAST                2 (quotient)
             26 LOAD_FAST                1 (b)
             28 LOAD_FAST                3 (remainder)
             30 CALL_FUNCTION            4
             32 CALL_FUNCTION            1
             34 POP_TOP
             36 LOAD_CONST               0 (None)
             38 RETURN_VALUE
```

Below, we can run the function to see the output.

```
display_divmod(23,5)
```

```
    23 = 4 (5) + 3
```

Notice that this function has no `return` line. The function terminates automatically at the end of its scope.

The function also uses Python's **string formatting**. This has changed between Python 2.x and 3.x, and this notebook uses Python 3.x syntax.

String formatting allows you to insert placeholders like `{}` within a string, and later fill those places with a list of things.

```
print("My favorite number is {}".format(17))  # The .format "method" substitutes 17 f
```

```
My favorite number is 17
```

```
print("{} + {} = {}".format(13,12,13+12))
```

```
13 + 12 = 25
```

The `format` command is an example of a **string method**. It has the effect of replacing all placeholders `{}` by the its inputs, in sequence. There is an intricate syntax for these placeholders, to allow one to match placeholders with values in different orders, and to format different kinds of values. Here is the official reference for string formatting in Python 3.x. We will only use the most basic features, exhibited below.

```
print ("The number {} comes before {}.".format(1,2)) # This should be familiar.
print ("The number {1} comes before {0}.".format(1,2)) # What happens?
print ("The number {1} comes before {1}.".format(1,2)) # Got it now?
```

```
The number 1 comes before 2.
The number 2 comes before 1.
The number 2 comes before 2.
```

By placing a number in the placeholder, like `{1}`, one can fill in the placeholders with the values in a different order, or repeat the same value. The format method takes multiple parameters, and they are numbered: parameter 0, parameter 1, parameter 2, etc.. So the placeholder `{1}` will be replaced by the second parameter (parameter 1). It's confusing at first, but Python almost always starts counting at zero.

```
print("pi is approximately {0}".format(3.14159265))
print("pi is approximately {0:f}".format(3.14159265)) # The "f" in "0:f" formats the
print("pi is approximately {0:0.3f}".format(3.14159265)) # Choose 3 digits of precisi
```

```
pi is approximately 3.14159265
pi is approximately 3.141593
pi is approximately 3.142
```

If you give some information about how the placeholder is being used, the format method will format things more nicely for printing. The placeholder `{0:f}` will be replaced by parameter 0, and it will be formatted in a way that is nice for floats (hence the `f`). Don't try formatting things outside of their type!

```
print("{:d} is a pretty big integer.".format(2**100)) # d is the formatting code for
print("{:f} is an integer, formatted like a float.".format(2**100))
print("{:f} is a float, of course.".format(1/7))
print("{:s} is a string.".format('Hi there!')) # s is the formatting code for strings
print("{:d} will give us an error message.".format(1/7))
```

```
1267650600228229401496703205376 is a pretty big integer.
1267650600228229401496703205376.000000 is an integer, formatted like a float.
0.142857 is a float, of course.
Hi there! is a string.
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-8-a243553941ff> in <module>()
      3 print("{:f} is a float, of course.".format(1/7))
      4 print("{:s} is a string.".format('Hi there!')) # s is the formatting
code for strings.
----> 5 print("{:d} will give us an error message.".format(1/7))

ValueError: Unknown format code 'd' for object of type 'float'
```

SEARCH STACK OVERFLOW

```
from math import sqrt  # Make sure the square root function is loaded.
print("The square root of {0:d} is about {1:f}.".format(1000, sqrt(1000)))
```

```
The square root of 1000 is about 31.622777.
```

Exercises

1. What are the signals of scope in Python?
2. Write a function called area_circle, which takes one argument radius. The function should return the area of the circle, as a floating point number. Then add one line to the function, using string formatting, so that it additionally prints a helpful sentence of the form "The area of a circle of radius 1.0 is 3.14159." (depending on the radius and the area it computes).
3. Write a function called factorial, which takes one argument called `n`. The function should return the factorian of `n` when `n` is a positive integer. Don't worry about what happens when

`n` is zero, or a bad input. Don't use recursion (if you know it) -- just use a for loop.

4. `format` is an example of a "string method". Another neat one is `replace`. Try `"Python".replace("yth","arag")` to see what it does.
5. Try the formatting codes `%` and `E` (instead of `f`) for a floating point number. What do they do?
6. Can you think of a reason you might want to have a function with *no* arguments?

Question 1

1. Scope refers to the names and variables within a function or code. When it comes to for loops, the colon and indentation are signals of scope.

Question 2

```
def area_circle(rad):
  pi = 3.14159
  area = pi * rad**2
  print("The area of a circle of radius {0:f} is {1:f}".format(rad,area))
  return area
```

```
area_circle(2)
```

```
    The area of a circle of radius 2.000000 is 12.566360
    12.56636
```

Question 3

```
def factorial(n):
  fac = 1
  for i in range(2, n + 1):
      fac = fac * i
  return fac
```

```
factorial(9)
```

```
    362880
```

Question 4

```
"Python".replace("yth", "arag")
```

```
'Paragon'
```

Question 5

```
print("{:%} is a float".format(1/7))
```

```
14.285714% is a float
```

```
print("{:E} is a float".format(1/7))
```

```
1.428571E-01 is a float
```

```
print("{:f} is a float".format(1/7))
```

```
0.142857 is a float
```

5. Above, we see the different formats for float numbers. It looks like the % symbol formats the decimal as a percentage. The E symbol uses scientific notation and of course f formats the number to float if it is not already formatted as such.

Question 6

6. My initial thought would be that an empty argument means that the parameters can be introduced inside the function. For example, if you left the argument space black and you prompted the user to input a value.

Control statements

It is important for a computer program to behave differently under different circumstances. The simplest control statements, `if` and its relative `else`, can be used to tell Python to carry out different actions depending on the value of a boolean variable. The following function exhibits the syntax.

```
def is_even(n):
    if n%2 == 0:
        print("{} is even.".format(n))
        return True
    else:
        print("{} is odd.".format(n))
        return False
```

```
is_even(17)
```

```
17 is odd.
False
```

```
is_even(1000)
```

```
1000 is even.
True
```

The broad syntax of the function should be familiar. We have created a function called `is_even` with one argument called `n`. The body of the function uses the **control statement** `if n%2 == 0:`. Recall that `n%2` gives the remainder after dividing `n` by `2`. Thus `n%2` is 0 or 1, depending on whether `n` is even or odd. Therefore the **boolean** `n%2 == 0` is `True` if `n` is even, and `False` if `n` is odd.

The next two lines (the first `print` and `return` statements) are within the **scope** of the `if <boolean>:` statement, as indicated by the colon and the indentation. The `if <boolean>:` statement tells the Python interpreter to perform the statements within the scope if the boolean is `True`, and to ignore the statements within the scope if the boolean is `False`.

Putting it together, we can analyze the code.

```
if n%2 == 0:
    print("{} is even.".format(n))
    return True
```

If `n` is even, then the Python interpreter will print a sentence of the form `n is even`. Then the interpreter will return (output) the value `True` and the function will terminate. If `n` is odd, the Python interpreter will ignore the two lines of scope.

Often we don't just want Python to *do nothing* when a condition is not satisfied. In the case above, we would rather Python tell us that the number is odd. The `else:` control statement tells Python what to do in case the `if <boolean>:` control statement receives a `False` boolean. We analyze the code

```
    else:
        print("{} is odd.".format(n))
        return False
```

The `print` and `return` commands are within the scope of the `else:` control statement. So when the `if` statement receives a false signal (the number `n` is odd), the program prints a sentence of [illegible]

The function `is_even` is a verbose, or "talkative" sort of function. Such a function is sometimes useful in an interactive setting, where the programmer wants to understand everything that's going on. But if the function had to be called a million times, the screen would fill with printed sentences! In practice, an efficient and silent function `is_even` might look like the following.

```
def is_even(n):
    return (n%2 == 0)
```

```
is_even(17)
```

A `for` loop and an `if` control statement, used together, allow us to carry out a **brute force** search. We can search for factors in order to check whether a number is prime. Or we can look for solutions to an equation until we find one.

One thing to note: the function below begins with a block of text between a triple-quote (three single-quotes when typing). That text is called a **docstring** and it is meant to document what the function does. Writing clear docstrings becomes more important as you write longer programs, collaborate with other programmers, and when you want to return months or years later to use a program again. There are different style conventions for docstrings; for example, here are Google's docstring conventions. We take a less formal approach.

```
def is_prime(n):
    '''
    Checks whether the argument n is a prime number.
    Uses a brute force search for factors between 1 and n.
    '''
    for j in range(2,n):  # the list of numbers 2,3,...,n-1.
        if n%j == 0:  # is n divisible by j?
            print("{} is a factor of {}.".format(j,n))
            return False
    return True
```

An important note: the `return` keyword **terminates** the function. So as soon as a factor is found, the function terminates and outputs `False`. If no factor is found, then the function execution survives past the loop, and the line `return True` is executed to terminate the function.

```
is_prime(91)
```

```
    7 is a factor of 91.
    False
```

```
is_prime(101)
```

```
    True
```

Try the `is_prime` function on bigger numbers -- try numbers with 4 digits, 5 digits, 6 digits. Where does it start to slow down? Do you get any errors when the numbers are large? Make sure to save your work first, just in case this crashes your computer!

```
is_prime(1234)
```

```
    2 is a factor of 1234.
    False
```

```
is_prime(12345)
```

```
    3 is a factor of 12345.
    False
```

```
is_prime(123456)
```

```
    2 is a factor of 123456.
    False
```

I could tell the runtime was progressively slowing for each added digit.

There are two limiting factors, which we study in more detail later. These are **time** and **space** (your computer's memory space). As the loop of `is_prime` goes on and on, it might take your computer a long time! If each step of the loop takes only a nanosecond (1 billionth of a second), the loop would take about a second when executing `is_prime(1000000001)`. If you tried `is_prime` on a much larger number, like `is_prime(2**101 - 1)`, the loop would take longer than the lifetime of the Earth.

The other issue that can arise is a problem with *space*. In Python 3.x, the `range(2,n)` cleverly *avoids* storing all the numbers between `2` and `n-1` in memory. It just remembers the endpoints, and how to proceed from one number to the next. In the older version, Python 2.x, the range command `range(2,n)` would have tried to store the entire list of numbers `[2,3,4,...,n-1]` in the memory of your computer. Your computer has some (4 or 8 or 16, perhaps) gigabytes of memory (RAM). A gigabyte is a billion bytes, and a byte is enough memory to store a number between 0 and 255. (More detail about this later!). So a gigabyte will not even hold a billion numbers. So our `is_prime` function would have led to memory problems in Python 2.x, but in Python 3.x we don't have to worry (for now) about space.

Exercises

1. Create a function `my_abs(x)` which outputs the absolute value of the argument `x`. (Note that Python already has a built-in `abs(x)` function).
2. Modify the `is_prime` function so that it prints a message `Number too big` and returns `None` if the input argument is bigger than one million. (Note that `None` is a Python reserved word. You can use the one-line statement `return None`.)
3. Write a Python function `thrarity` which takes an argument `n`, and outputs the string `threeven` if `n` is a multiple of three, or `throdd` is `n` is one more than a multiple of three, or `thrugly` if `n` is one less than a multiple of three. Example: `thrarity(31)` should output `throdd` and `thrarity(44)` should output `thrugly`. Hint: study the `if` / `elif` syntax at the official Python tutorial
4. Write a Python function `sum_of_squares(n)` which finds and prints a pair of natural numbers x, y, such that $x^2 + y^2 = n$. The function should use a brute force search and return `None` if no such pair of numbers x, y exists. Explore which natural numbers can be expressed as sums of two squares... hint: look at prime numbers first!
5. Write a function `gamma(n)` which takes a positive integer n as input, and outputs the difference between the harmonic sum $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ and the natural logarithm $\log(n)$. Use numpy to compute the logarithm, by using the command `from numpy import log` in a separate cell. Approximate $\gamma(n)$ as $n \to \infty$. How large does n need to be to get five digits of precision on this limit? Can you prove that the limit $\lim_{n\to\infty} \gamma(n)$ exists?

Question 1

```
def my_abs(x):
  if x < 0:
    return -1*x
  else:
    return x


my_abs(-20)
```

```
20
```

Question 2

```
def is_prime(n):
    '''
    Checks whether the argument n is a prime number.
    Uses a brute force search for factors between 1 and n.
    '''
    if n > 1000000:
      print("Number too big")
      return None
    for j in range(2,n):  # the list of numbers 2,3,...,n-1.
        if n%j == 0:  # is n divisible by j?
            print("{} is a factor of {}.".format(j,n))
            return False
    return True
```

```
is_prime(4)
```

```
2 is a factor of 4.
False
```

Question 3

```
def thrarity(n):
  if n%3 == 0:
    print("{} is threeven".format(n))
  elif n%3 == 1:
    print("{} is throdd".format(n))
  elif n%3 == 2:
    print("{} is thrugly".format(n))
```

```
thrarity(44)
```

```
44 is thrugly
```

Question 4

```
def sum_of_squares(n):
  for x in range(1, n+1):
    for y in range(1, n+1):
      if (x*x + y*y == n):
        print("{0}^2 + {1}^2 = {2}".format(x,y,n))
      y = y + 1
    x = x + 1
    return None
```

```
sum_of_squares(52)
```

```
    4^2 + 6^2 = 52
    6^2 + 4^2 = 52
```

Question 5

```
from numpy import log
```

```
def gamma(n):
  for i in range(0,n):
    n = n + 1/(i+1)
  print("harmonic: {}".format(n))
  gm = n - log(n)
  return gm
```

```
gamma(50)
```

```
    harmonic: 54.49920533832947
    50.501019217714656
```

It seems like gamma(n) diverges as n approaches infinity. Seems like n just has to be greater than 0 to evaluate gamma.

Handling errors by raising exceptions.

In the previous batch of exercises, we tried to modify functions to be a bit more intelligent -- identifying when numbers were "too big" for example. There's a professional way to handle these situations, by raising *exceptions*. Here is the official documentation on errors and exceptions. We will focus on raising exceptions to catch "bad inputs" to functions. Let's revisit our `is_even` function.

```
def is_even(n):
    return (n%2 == 0)
```

```
is_even(3.14)  # What will this do?
```

```
    3.14 is odd.
    False
```

```
3.14%2  # Well, this explains it!
```

```
    1.1400000000000001
```

Although the output of `is_even(3.14)` might be what you want, a smarter function might let the user know that 3.14 should not be input into `is_even`. We commonly ask whether *integers* are even or odd; if a non-integer ends up input to `is_even`, it might be a sign of a bug elsewhere. One possibility is to modify the function by manually printing an error message.

```
def is_even(n):
    if type(n) == int:
        return (n%2 == 0)
    else:
        print("Bad input!  Please input integers only.")
        return None
```

```
is_even(4)
```

```
    True
```

```
is_even(3.14)
```

```
    Bad input!  Please input integers only.
```

```
print(is_even(3.14))
```

```
    Bad input!  Please input integers only.
    None
```

This behavior is a bit better. The output of the function is neither True nor False, when a non-integer is input. Instead, the smarter function outputs `None`, which is exactly what it sounds like.

```
type(None) # A zen command.
```

Instead of manually using a print command and returning None, we can use Python's built-in `exception` class. Raising exceptions is the Pythonic way of catching errors, and this will make things smoother in the long term. Here's a new and safe `is_even` function.

```
def is_even(n):
    if type(n) == int:
        return (n%2 == 0)
    else:
        raise TypeError('Only integers can be even or odd.')
```

```
is_even(3)
```

```
False
```

```
is_even(3.14)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-24ec50760f46> in <module>()
----> 1 is_even(3.14)

<ipython-input-8-965faba9098f> in is_even(n)
      3         return (n%2 == 0)
      4     else:
----> 5         raise TypeError('Only integers can be even or odd.')

TypeError: Only integers can be even or odd.
```

SEARCH STACK OVERFLOW

Instead of manually printing the error message and returning `None`, we have raised a `TypeError`. This gives information about the kind of error, and a custom error message is displayed at the end. Type errors are meant for situations where a variable belongs to the wrong type. `TypeError` is just one kind of "exception" -- the full built-in hierarchy of exceptions can be found in the official Python documentation.

Another kind of exception is the `ValueError`. It seems similar to `TypeError` at first, but `ValueError` is meant to catch an input that has a "bad" value, even if it is the right type. For example, here is a square root function that only works with positive input. It should raise an exception (error message) when a negative number is input. Both positive and negative numbers can be represented as floats, so the error doesn't represent the *wrong type*. The error represents a *bad value*.

```
def sqrt(x):
    '''
    Estimates the square root of a positive number x.
    '''
    if x < 0:
        raise ValueError('Cannot approximate square root of negative numbers.')
    guess= x/2 # A decent place to start
    while True: # A dangerous loop!  See next section...
        new_guess = 0.5 * (guess + x/guess)
        if abs(new_guess - guess) < .000000001: # close enough!
            return new_guess
        guess = new_guess
```

```
sqrt(3) # This should be ok.
```

```
1.7320508075688772
```

```
sqrt(-3)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-8be5c01c52c0> in <module>()
----> 1 sqrt(-3)

<ipython-input-19-39991b3bf836> in sqrt(x)
      4     '''
      5     if x < 0:
----> 6         raise ValueError('Cannot approximate square root of negative
numbers.')
      7     guess= x/2 # A decent place to start
      8     while True: # A dangerous loop!  See next section...

ValueError: Cannot approximate square root of negative numbers.
```

SEARCH STACK OVERFLOW

By raising the `ValueError`, we have avoided an endless loop... the sort of problem that crashes computers! If you know that your function is only meant for certain kinds of inputs, it is best to catch errors by raising exceptions.

Exercises

1. There's a special exception called `ZeroDivisionError`. When do you think this occurs? Try to make it occur! Can you think of a time when you might want to raise this exception yourself?
2. Make the `is_prime` function safer by raising a `TypeError` or a `ValueError` when a "bad" input occurs.

Question 1

```
1/0
```

```
---------------------------------------------------------------------------
```

1. As one can see, the ZeroDivisionError occurs when numbers are divided by zero. Obviously raising a TypeError or ValueError would be the best way to prevent the occurence, which would most likely be due to code in which the input of divisors could possibly be zero. For example, any function where time could be inputted as a divisor, one might want to raise a ValueError.

Question 2

```
def is_prime(n):
    '''
    Checks whether the argument n is a prime number.
    Uses a brute force search for factors between 1 and n.
    '''
    if n > 1000000:
      raise ValueError("Number too big")
    for j in range(2,n):  # the list of numbers 2,3,...,n-1.
        if n%j == 0:  # is n divisible by j?
            print("{} is a factor of {}.".format(j,n))
            return False
    return True
```

```
is_prime(2000000)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-24-f5e5954b6fe8> in <module>()
----> 1 is_prime(2000000)

<ipython-input-23-609e3b942239> in is_prime(n)
      5     '''
      6     if n > 1000000:
----> 7       raise ValueError("Number too big")
      8     for j in range(2,n):  # the list of numbers 2,3,...,n-1.
      9         if n%j == 0:  # is n divisible by j?

ValueError: Number too big
```

SEARCH STACK OVERFLOW

While loops and implementation of the Eucidean algorithm

We *almost* have all the tools we need to implement the Euclidean algorithm. The last tool we will need is the **while loop**. We have seen the *for loop* already, which is very useful for iterating over a range of numbers. The Euclidean algorithm involves repetition, but there is no way to know in advance how many steps it will take. The while loop allows us to repeat a process as long as a boolean value (sometimes called a **flag**) is True. The following countdown example illustrates the structure of a while loop

```
def countdown(n):
    current_value = n
    while current_value > 0:  # The condition (current_value > 0) is checked before e
        print(current_value)
        current_value = current_value - 1
    print("Blastoff!")


countdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
Blastoff!
```

The while loop syntax begins with `while <boolean>:` and the following indented lines comprise the scope of the loop. If the boolean is `True`, then the scope of the loop is executed. If the boolean is `True` again afterwards, then the scope of the loop is executed again. And again and again and so on.

This can be a **dangerous process**! For example, what would happen if you made a little typo and the last line of the while loop read `current_value = current_value + 1`? The numbers would increase and increase... and the boolean `current_value > 0` would **always** be `True`. Therefore the loop would never end. Bigger and bigger numbers would scroll down your computer screen.

You might panic under such a circumstance, and maybe turn your computer off to stop the loop. Here is some advice for when you get stuck in such a neverending loop, and you're using Google Colab.

1. Back up your work often. When you're programming, make sure everything else is saved just in case.
2. Save your programming work (use "Save a copy in GitHub" and/or "Save a copy in Drive") often, especially before running a cell with a loop for the first time.

3. If you *do* get stuck in a neverending loop, click on "Runtime... Interrupt execution". This will often unstick the loop and allow you to pick up where you left off.

Now, if you're feeling brave, save your work, change the while loop so that it never ends, and try to recover where you left off. But be aware that this could cause your computer to freeze or behave erratically, crashing your browser, etc. Don't panic... it probably won't break your computer permanently.

The neverending loop causes two problems here. One is with the computer processor, which will be essentially spinning its wheels. This is called busy waiting, and the computer will essentially be busy waiting forever. The other problem is that your loop is printing more and more lines of text into the notebook. This could easily crash your web browser, which is trying to store and display zillions

The Euclidean algorithm with a while loop

The **Euclidean Algorithm** is a process of repeated division with remainder. Beginning with two integers `a` (dividend) and `b` (divisor), one computes quotient `q` and remainder `q` to express `a = qb + r`. Then `b` becomes the dividend and `r` becomes the divisor, and one repeats. The repetition continues, and the **last nonzero** remainder is the greatest common divisor of `a` and `b`.

We implement the Euclidean algorithm in a few variations. The first will be a verbose version, to show the user what happens at every step. We use a while loop to take care of the repetition.

```
def Euclidean_algorithm(a,b):
    dividend = a
    divisor = b
    while divisor != 0:   # Recall that != means "is not equal to".
        quotient = dividend // divisor
        remainder = dividend % divisor
        print("{} = {} ({}) + {}".format(dividend, quotient, divisor, remainder))
        dividend = divisor
        divisor = remainder
```

```
Euclidean_algorithm(133, 58)
```

```
133 = 2 (58) + 17
58 = 3 (17) + 7
17 = 2 (7) + 3
7 = 2 (3) + 1
3 = 3 (1) + 0
```

```
Euclidean_algorithm(1312331323, 58123123)
```

```
1312331323 = 22 (58123123) + 33622617
```

```
58123123 = 1 (33622617) + 24500506
33622617 = 1 (24500506) + 9122111
24500506 = 2 (9122111) + 6256284
9122111 = 1 (6256284) + 2865827
6256284 = 2 (2865827) + 524630
2865827 = 5 (524630) + 242677
524630 = 2 (242677) + 39276
242677 = 6 (39276) + 7021
39276 = 5 (7021) + 4171
7021 = 1 (4171) + 2850
4171 = 1 (2850) + 1321
2850 = 2 (1321) + 208
1321 = 6 (208) + 73
208 = 2 (73) + 62
73 = 1 (62) + 11
62 = 5 (11) + 7
11 = 1 (7) + 4
7 = 1 (4) + 3
4 = 1 (3) + 1
3 = 3 (1) + 0
```

This is excellent if we want to know every step of the Euclidean algorithm. If we just want to know the GCD of two numbers, we can be less verbose. We carefully return the last nonzero remainder after the while loop is concluded. This last nonzero remainder becomes the divisor when the remainder becomes zero, and then it would become the dividend in the next (unprinted) line. That is why we return the (absolute value) of the dividend after the loop is concluded. You might insert a line at the end of the loop, like `print(dividend, divisor, remainder)` to help you track the variables.

```
def GCD(a,b):
    dividend = a # The first dividend is a.
    divisor = b # The first divisor is b.
    while divisor != 0:   # Recall that != means "not equal to".
        quotient = dividend // divisor
        remainder = dividend % divisor
        dividend = divisor
        divisor = remainder
    return abs(dividend)  #  abs() is used, since we like our GCDs to be positive.
```

Note that the `return dividend` statement occurs *after* the scope of the while loop. So as soon as the *divisor* variable equals zero, the funtion `GCD` returns the *dividend* variable and terminates.

```
GCD(111,27)
```

```
3
```

```
GCD(111,-27)
```

```
3
```

We can refine our code in a few ways. First, note that the `quotient` variable is never used! It was nice in the verbose version of the Euclidean algorithm, but plays no role in finding the GCD. Our refined code reads

```
def GCD(a,b):
    dividend = a
    divisor = b
    while divisor != 0:   # Recall that != means "not equal to".
        remainder = dividend % divisor
        dividend = divisor
        divisor = remainder
    return abs(dividend)
```

Now there are two slick Python tricks we can use to shorten the code. The first is called **multiple assignment**. It is possible to set the values of two variables in a single line of code, with a syntax like below.

```
x,y = 2,3  # Sets x to 2 and y to 3.
```

This is particular useful for self-referential assignments, because as for ordinary assignment, the right side is evaluated first and then bound to the variables on the left side. For example, after the line above, try the line below. Use print statements to see what the values of the variables are afterwards!

This would switch the variable's values.

```
x,y = y,x #  Guess what this does!
```

```
print("x =",x) # One could use "x = {}".format(x) too.
print("y =",y)
```

```
x = 3
y = 2
```

Now we can use multiple assignment to turn three lines of code into one line of code. For the `remainder` variable is only used temporarily before its value is given to the `divisor` variable. Using multiple assignment, the three lines

```
    remainder = dividend % divisor
    dividend = divisor
    divisor = remainder
```

can be written in one line,

```
    dividend, divisor = divisor, dividend % divisor # Evaluations on the right occur before
```

Our newly shortened GCD function looks like this.

```
def GCD(a,b):
    dividend = a
    divisor = b
    while divisor != 0:   # Recall that != means "not equal to".
        dividend, divisor = divisor, dividend % divisor
    return abs(dividend)
```

The next trick involves the while loop. The usual syntax has the form `while <boolean>:`. But if `while` is followed by a numerical type, e.g. `while <int>:`, then the scope of the while loop will execute as long as the number is nonzero! Therefore, the line

```
while divisor != 0:
```

can be replaced by the shorter line

```
while divisor:
```

This is truly a trick. It probably won't speed anything up, and it does not make your program easier to read for beginners. So use it if you prefer communicating with experienced Python programmers! Here is the whole function again.

```
def GCD(a,b):
    dividend = a
    divisor = b
    while divisor:   # Executes the scope if divisor is nonzero.
        dividend, divisor = divisor, dividend % divisor
    return abs(dividend)
```

The next optimization is a bit more dangerous for beginners, but it works here. In general, it can be dangerous to operate directly on the arguments to a function. But in this setting, it is safe, and makes no real difference to the Python interpreter. Instead of creating new variables called `dividend` and `divisor`, one can manipulate `a` and `b` directly within the function. If you do this, the GCD function can be shortened to the following.

```
def GCD(a,b):
    while b:   # I.e., while b != 0.
        a, b = b, a % b
    return abs(a)
```

```
GCD(1234,5678)
```

```
2
```

This code is essentially optimal, if one wishes to execute the Euclidean algorithm to find the GCD of two integers. It almost matches the GCD code in a standard Python library. It might be slightly faster than our original code -- but there is a tradeoff here between execution speed and readability of code. In this and the following lessons, we often optimize enough for everyday purposes, but not so much that readability is lost.

Exercises and explorations

1. Modify the `is_prime` function by using a while loop instead of `for j in range(2,n):`. Hint: the function should contain the lines `j = 2` and `while j < n:` and `j = j + 1` in various places. Why might this be an improvement from the for loop? Can you look for factors within a smaller range?
2. Modify the `Euclidean_algorithm` function to create a function which returns the *number of steps* that the Euclidean algorithm requires, i.e., the number of divisions-with-remainder. How does the number of steps compare to the size of the input numbers?
3. When a and b are integers, $GCD(a, b) \cdot LCM(a, b) = ab$. Use this fact to write an LCM-function. Try to make your function output only integers (not floats) and behave in a good way even if a, b are zero.
4. How does the `GCD(a,b)` function behave when `a` and/or `b` are zero or negative? Is this good?
5. Challenge: Write a function `approximate_e(n)` which approximates e with a maximum error of 10^{-n}. (You can assume $n < 1000$ if necessary.) Try a while loop and `import mpmath`.

Back up your work often!

Question 1

```
def is_prime(n):
    '''
    Checks whether the argument n is a prime number.
    Uses a brute force search for factors between 1 and n.
    '''
    if n > 1000000:
      print("Number too big")
      return None
    j = 2
    while j < n:
        if n%j == 0:  # is n divisible by j?
            print("{} is a factor of {}.".format(j,n))
            return False
        j = j + 1
    return True
```

```
is_prime(150)
```

```
2 is a factor of 150.
False
```

1. I didn't notice a runtime improvement, so I'm guessing modifying to a while loop has more to do with being able to widen the scope of the function. We know that for loops are used when the number of iterations needed is known, whereas a while loop can break under a condition, so it would be easier to manipulate a while loop by adding a condition.

Question 2

```
def Euclidean_algorithm(a,b):
    dividend = a
    divisor = b
    count = 0
    while divisor != 0:   # Recall that != means "is not equal to".
        quotient = dividend // divisor
        remainder = dividend % divisor
        dividend = divisor
        divisor = remainder
        count = count + 1
    print("Number of steps:")
    return count
```

```
Euclidean_algorithm(1007, 1589)
```

```
Number of steps:
12
```

2. It seems the larger the integers inputted into this function, the larger the number of steps unless the numbers share big common divisors.

Question 3

```
def lcm(a, b):
    return (a*b) // GCD(a, b)
```

```
lcm(70, 100)
```

```
700
```

Question 4

```
GCD(-4,10)
```

```
2
```

```
GCD(-7, -70)
```

```
7
```

```
GCD(0,21)
```

```
21
```

4. Seems as though negative or zero values of a and/or b do not affect the accuracy of the GCD function.

Question 5

```
def approximate_e(n):
  while n != 0:
  while n != 0:
    return (1 + (1/n))**n


approximate_e(100000)

     2.7182682371922975
```