
SUNCORP
Python School
Lesson 3
~~13 April 2017~~ 28 April 2017

PREVIOUSLY ON PYTHON SCHOOL

WE LOOKED AT

FLOW CONTROL!

for loops

while loops

ifelse statements

Statements that allow us to **control** processes and **repeat** them many times

allowing

AUTOMATION

while automating our code in this way is powerful

---

for our code to be effective it needs to be

TRACTABLE

COHERENT

SIMPLE

FOR EXAMPLE

writing this out every time I need to import a file would be repetitive

```
# need to load in one CSV file
with open("my_file.csv") as file:
    dataset1 = []
    for (i, line) in enumerate(file):
        data_row = line.strip().split(",")
        if i > 0:
            dataset1.append(data_row)
        else:
            header = data_row

# then do it again for a tab-delimited file
with open("my_file2.txt") as file2:
    dataset2 = []
    for (i, line) in enumerate(file2):
        data_row = line.strip().split("\t")
        if i > 0:
            dataset2.append(data_row)
        else:
            header2 = data_row
```

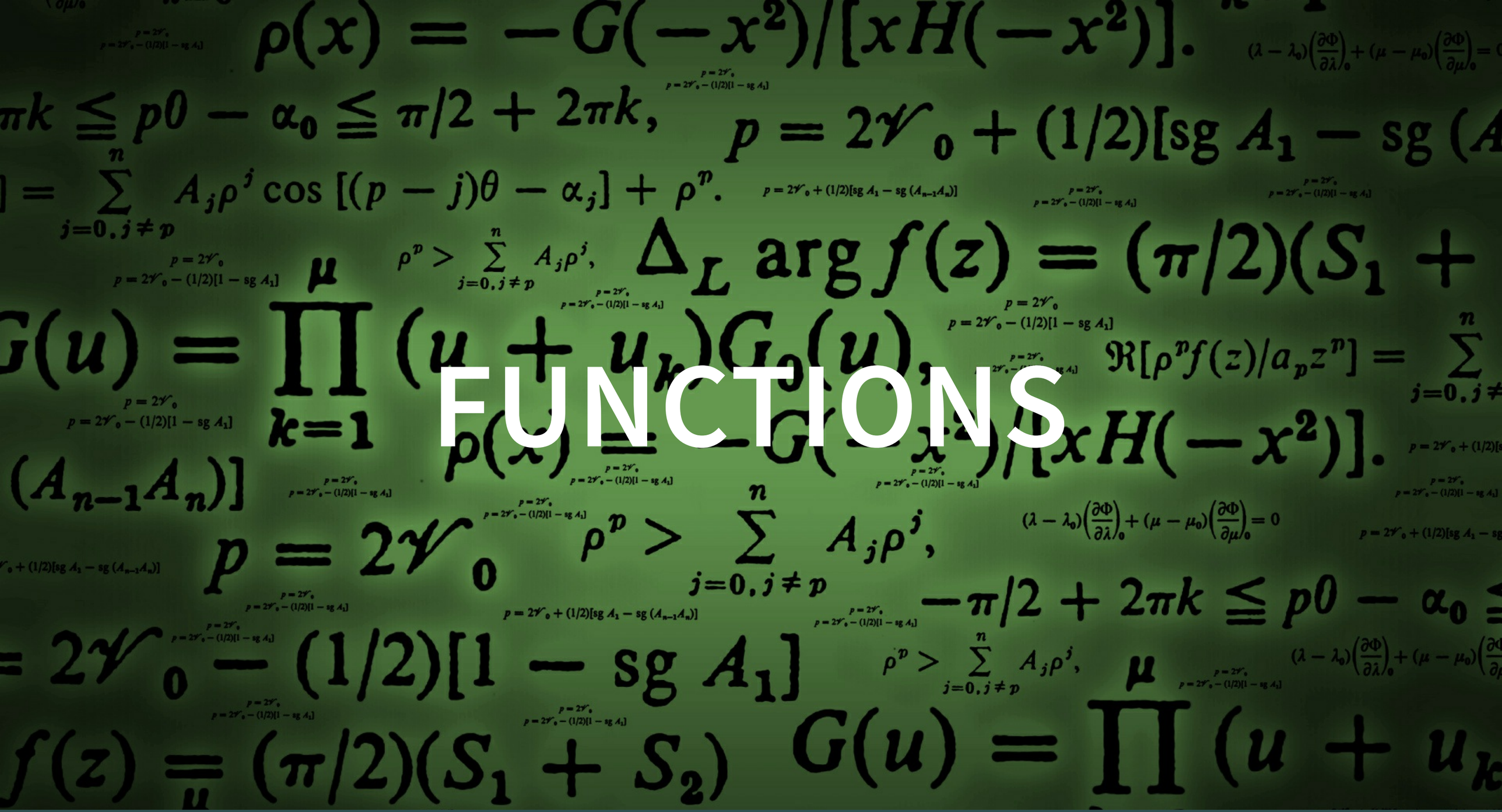
it also makes it difficult to see what is happening

I could throw this into another loop, with some ifelse statements

but it would still be complicated and hard to read

what if i could encapsulate this process...?

and just call it when I need it



FUNCTIONS

WHAT IS A FUNCTION?

anything that operates on an input

to determine an output

FOR EXAMPLE

the area of a circle is a **function** of its radius

$$A = \pi r^2$$

or

$$f(x) = \pi x^2$$

so why do we like them?

functions allow for

BREVITY

in describing one or many processes

FOR EXAMPLE

say I have an equation for calcluating compound interest

$$P' = P\left(1 + \frac{r}{n}\right)^{nt}$$

where P = principal sum , P' = new sum , r = nominal interest rate , n = compounding frequency , t = time of investment

it would laborious to write this out every time I have some different permutation of inputs

rather I would be better defining it like so

$$f(P, r, n, t) = P\left(1 + \frac{r}{n}\right)^{nt}$$

so that we can just write

$$P' = f(P, r, n, c)$$

this mathematical abstraction

to **encapsulate** and execute any arbitrary process at will

APPLIES TO PROGRAMMING AS WELL

functions are expressed in Python as follows:

```
def compound_interest(P, r, n, t):
    """
    calculates compound interest
    """

    P_new = P*(1 + r/n)**(n*t)

    return P_new
```

functions are expressed in Python as follows:

```
def compound_interest(P, r, n, t):
    """
    calculates compound interest
    """

    P_new = P*(1 + r/n)**(n*t)

    return P_new
```

use `def` to tell Python we're creating a function

functions are expressed in Python as follows:

```
def compound_interest(P, r, n, t):
    """
    calculates compound interest
    """

    P_new = P*(1 + r/n)**(n*t)

    return P_new
```

we also have to give the function a name, for reference

functions are expressed in Python as follows:

```
def compound_interest(P, r, n, t):
    """
    calculates compound interest
    """

    P_new = P*(1 + r/n)**(n*t)

    return P_new
```

and tell it what **input variables** it accepts

functions are expressed in Python as follows:

```
def compound_interest(P, r, n, t):
    """
    calculates compound interest
    """

    P_new = P*(1 + r/n)**(n*t)

    return P_new
```

the input(s) are then operated on in some way

functions are expressed in Python as follows:

```
def compound_interest(P, r, n, t):
    """
    calculates compound interest
    """

    P_new = P*(1 + r/n)**(n*t)

    return P_new
```

to determine an outcome, which is **returned** to the user

functions are expressed in Python as follows:

```
def compound_interest(P, r, n, t):
    """
    calculates compound interest
    """

    P_new = P*(1 + r/n)**(n*t)

    return P_new
```

lastly, we define what the function does for reference

variables in functions can only affect that specific process

∴ outside the function, they hold no purpose

...

this leads on to our next concept

SCOPE

a variable's visabilty and ability to affect the program

when we talk about scope, variables come in two contexts

LOCAL & GLOBAL

let's look at both of these

LOCAL

VARIABLES

Rhys Whitley • Python School • 30 March 2017

local variables are those defined within a function

```
def quadratic(x):
    """
    determines the square of a number
    """
    a = 2
    y = a*x**2

    return y
```

exist **only within** the scope → code block of the function

once the function ends, the variable is destroyed

FOR EXAMPLE

```
def quadratic(x):
    """
    determines the square of a number
    """
    a = 2
    y = a*x**2

    return y
```

any reference to the a variable outside this function

```
In : print(a)
Out: NameError: name 'a' is not defined
```

will fail → as it exists **ONLY** within the function

this is apparent when we have a function that calls another

```
def quadratic(x):
    """returns the square"""
    y = x**2 + shift

    return y
```

```
def shifted_quadratic(x):
    """
    shifts the square along x
    """
    shift = 10
    ans = quadratic(x)

    return ans
```

`quadratic()` will **fail**
because it has no idea what
`shift` is!

`shift` belongs to
`shifted_quadratic()`
which only IT can see it

defined separately, the functions do **NOT** see each other

```
def quadratic(x):
    """returns the square"""
    y = x**2 + shift

    return y
```

```
def shifted_quadratic(x):
    """
    shifts the square along x
    """
    shift = 10
    ans = quadratic(x)

    return ans
```

but we can fix this code so that they don't need to

we just change `quadratic()` to accept an extra variable → `shift`

defined separately, the functions do **NOT** see each other

```
def quadratic(x, shift):
    """returns the square"""
    y = x**2 + shift

    return y
```

```
def shifted_quadratic(x):
    """
    shifts the square along x
    """
    shift = 10
    ans = quadratic(x, shift)

    return ans
```

but we can fix this code so that they don't need to

we just change `quadratic()` to accept an extra variable → `shift`

`quadratic()` now accepts both `x` and `shift` as inputs

GLOBAL

VARIABLES

Rhys Whitley • Python School • 30 March 2017

a variable that can be accessed by any part of the program

```
glob_shift = 10

def quadratic(x):
    """
    determines the square of a number
    """
    y = x**2 + glob_shift

    return y
```

here `glob_shift` is seen by `quadratic()` as it has full scope

declared outside any function / top level

SO WHAT IS THE BEST WAY TO DEFINE YOUR FUNCTION?

THERE IS NO RULE...BUT

best to explicitly pass all variables the function will need

---

if it needs x, y, and z

```
def operation():
    """adds 3 together"""
    axis = x + y + z

    return axis
```

then pass it x, y, and z

```
def operation(x, y, z):
    """adds 3 together"""
    axis = x + y + z

    return axis
```

you then directly control what goes in and out of your functions → process

NOW LET'S LOOK AT FUNCTIONS IN ACTION

HERE'S A TASK

imagine I want to sum all the numbers from $0 \rightarrow N$

$$y = \sum_{i=0}^{N} i$$

how would I go about it?

I COULD CODE IT AS

...which will work...

```
sum_i = 0
N = 50

for i in range(N + 1):

    sum_i += i

print(sum_i)
```

but what happens when I want to calculate many different values of N?

it would be inefficient to copy and paste this code every single time!

it would be more efficient to make this a function

```
def iter_sum(N):
    """
    iteratively sums numbers from 1 to N
    """
    sum_i = 0

    for i in range(N + 1):
        sum_i += i

    return sum_i
```

I can now call this whenever I want an answer

```
In : iter_sum(50)
Out: 1275
```

and do some cool things with it!

```
end = 50

for n in range(end + 1):

    csum = iter_sum(n)

    print(csum)
```

like calculate the cumulative sum for $M = 0 \rightarrow 10$

```
0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55
```

heck, let's make **this process** a function!

all our new function needs to do is call → `iter_sum()`

```
def cumul_sum(end):
    """calculates the cumulative sum for numbers 0 to M"""
    sum_list = []

    for n in range(end + 1):

        csum = iter_sum(n)

        sum_list.append(csum)

    return sum_list
```

and I can now deploy this in my code wherever I want

```
In : cumul_sum(10)
Out: [0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
```

functions fulfill the need to efficiently repeat tasks

a lot of the functions you create, may be useful in other programs/tasks

→ which introduces our next concept...

MODULES

Rhys Whitley • Python School • 30 March 2017

modules allow you to store functions separately from the main body of your code (or notebook)

sum_functions.py

```
def iter_sum(end):
    """sums the numbers 1 to N :: iter"""
    sum_i = 0

    for i in range(int(end) + 1):
        sum_i += i

    return sum_i

def tri_sum(end):
    """
    sum of numbers from 1 to N :: triangle
    """
    t_sum = end/2*(end + 1)

    return int(t_sum)

def cumul_sum(end, f=iter_sum):
    """calculates the cumulative sum for numbers 0 to M"""
    sum_list = []
```

IMPORT

`import` ... loads your *custom functions* as a **module** into a notebook (or program) like so

```
import sum_functions
```

filename = **module** name

```
sum_functions.iter_sum(10)
```

and any functions in the file are referred to as **methods**

and are accessed using **dot notation**

your module/filename is likely to be quite long, but Python allows us to abbreviate to a shorter reference name

```
import sum_functions as sf
```

using the `as` statement after the `import`

```
sf.iter_sum(10)
```

we can then call our specific method(s) as per normal

→ much cleaner ←

lastly, we don't always have to import the entire file!

maybe we just want the `iter_sum()` function

```
from sum_functions import iter_sum
```

by using `from` we can easily do this

```
iter_sum(10)
```

then call the function as per normal

storing your functions in modules is an efficient

& RECOMMENDED

way of building your program

HOWEVER

a LOT of extra modules are already available to you

EXTENDED PYTHON

Rhys Whitley • Python School • 30 March 2017

Python already has impressive standard library

that has been extended by the community, to provide modules for almost anything you can think of!

↓

linear algebra	scientific plotting	image recognition	geospatial analysis
statistics	machine learning	Bayesian inference	web scraping
HTML, XML, JSON	network analysis	database management	App. development

HERE ARE SOME EXAMPLES

want to do some basic math and geometry?

`math` has got you covered

```
from math import pi

r = 5
# area of a circle
area = pi*r**2
```

```
from math import sqrt

a = 3
b = 4
# hypotenuse
c = sqrt(a**2 + b**2)
```

what about some linear algebra?

---

hello numpy!

```
import numpy as np

# define two vectors
x_vec = np.arange(1, 8)[:, None]
y_vec = np.arange(1, 6)[:, None]

# and determine their dot product
matrix = np.dot(x_vec, y_vec.T)
```

```
Out: array([[ 1,  2,  3,  4,  5],
            [ 2,  4,  6,  8, 10],
            [ 3,  6,  9, 12, 15],
            [ 4,  8, 12, 16, 20],
            [ 5, 10, 15, 20, 25],
            [ 6, 12, 18, 24, 30],
            [ 7, 14, 21, 28, 35]])
```

need to plot something?

---

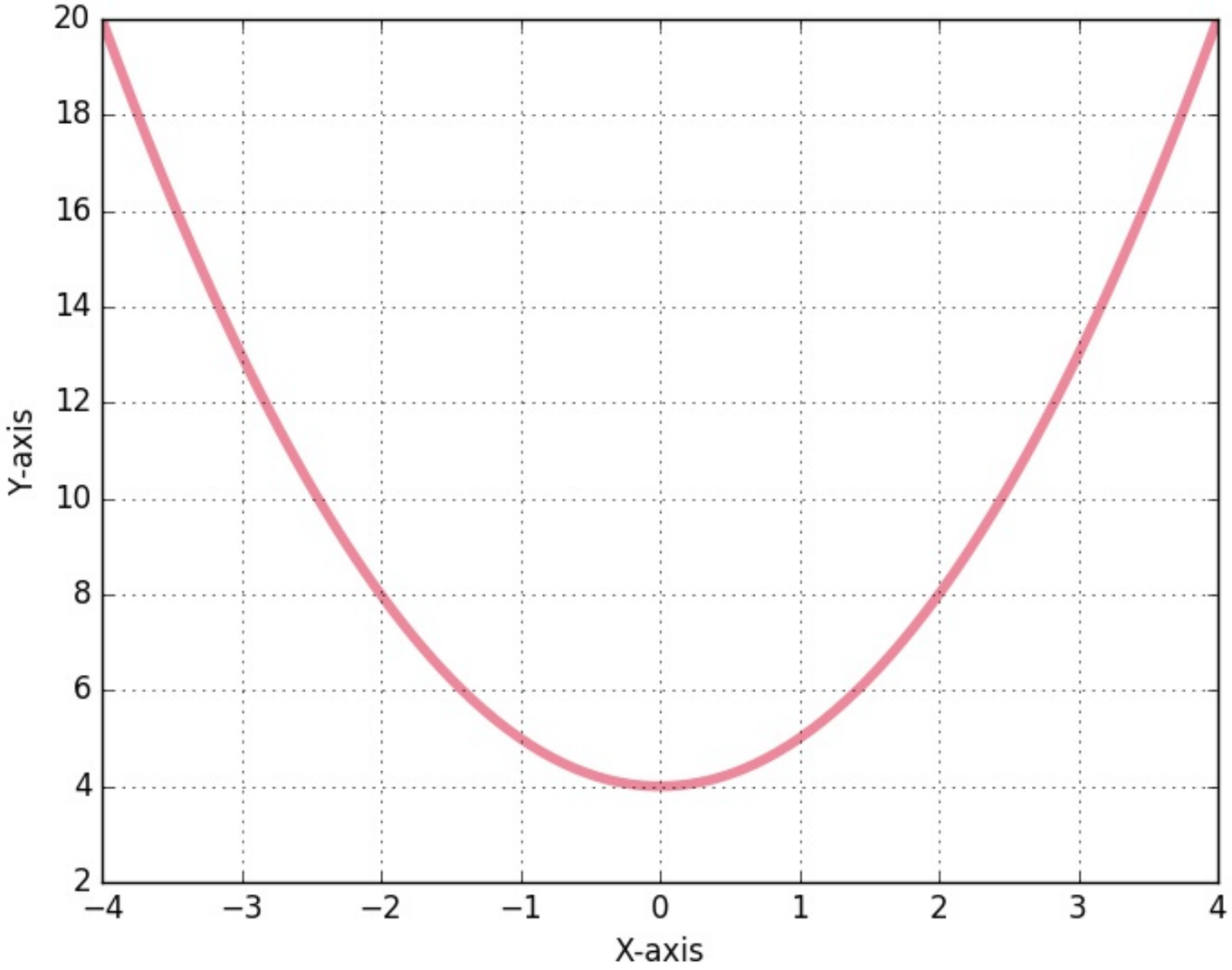
our good friend `matplotlib`

```
import matplotlib.pyplot as plt

def fquad(x):
    return 3+x**2 + 1

xs = list(range(-4, 5, 1))
ys = list(map(fquad, xs))

plt.plot(xs, ys, '-', color='crimson', alpha=0.5)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
```


20
18
16
14
12
10
8
6
4
2
Y-axis
−4
−3
−2
−1
0
1
2
3
4
X-axis

AND MANY MANY MORE

www.python.org

A FEW MORE THINGS TO BE AWARE OF

THE ORDER

of passed variables is important

```
def linear_model(x, a, b):
    """describes a linear relationship"""
    y = a * x + b
    return y
```

✔ correct order

```
In : linear_model(1, 2, 3):
Out: 5
```

✗ wrong order

```
In : linear_model(3, 2, 1):
Out: 7
```

HOWEVER

you can explicitly tell your function what you're passing

```
In : linear_model(b=3, a=2, x=1)
Out: 5
```

although I'd recommend you pass arguments in the order your function is expecting them

functions can also have

DEFAULT VALUES

```
def linear_model(x, a=1, b=0):
    """describes a linear relationship"""
    y = a * x + b
    return y
```

but **MUST** be defined at the tail end of passed arguments

```
In : linear_model(9):
Out: 9
```

```
In : linear_model(9, b=2):
Out: 11
```

```
In : linear_model(9, 2):
Out: 18
```

```
In : linear_model(9, 2, 2):
Out: 20
```

FUNCTIONS CAN ACCEPT AND RETURN

ANYTHING

numbers, strings, lists, dicts, even other functions

SEQUENCES

here we accept a **number** and return a dict

```
def sphere_object(radius):
    """
    determine sphere attributes
    """
    # define pi (I would usually import this)
    pi = 3.14159265359

    # calculate volume
    volume = 4/3*pi*radius**3
    # calculate surface area
    surf_area = 4*pi*radius**2

    # return value
    return {'volume': volume, 'surf_area': surf_area}
```

SEQUENCES

or if you prefer...a tuple

```
def sphere_object(radius):
    """
    determine sphere attributes
    """
    # define pi (I would usually import this)
    pi = 3.14159265359

    # calculate volume
    volume = 4/3*pi*radius**3
    # calculate surface area
    surf_area = 4*pi*radius**2

    # return value
    return volume, surf_area
```

we can also return incomplete functions with

CLOSURES

```
def linear_model(a, b):
    """y = a*x + b"""

    def x_response(x):
        return a*x + b

    return x_response

# define two models
model_1 = linear_model(3, 0)
model_2 = linear_model(1, 6)
```

```
In : model_1(4)
Out: 12

In : model_2(4)
Out: 10
```

ARGUMENT VERSATILITY

Python functions can accept any number of arguments

---

which allows us to exploit a behaviour called

*args

and

**kwargs

we can define our functions to accept any number of

ARBITRARY ARGUMENTS

---

where they can be just values → `*args`

or key-value pairs → `**kwargs`

*args

say I want a function that adds any numbers I pass to it

```
def add_me(*numbers):
    """adds all numbers together"""

    total = 0
    for num in numbers:
        total += num

    return total
```

I can do that quite easily

```
In : add_me(2, 4, 5, 6)
Out: 17

In : add_me(12)
Out: 12
```

HOT POTATO!

`*args` is also quite useful at passing arguments through a chain of functions

```
def predict_linear(x_series, a, b):
    """
    predicts Y for a series of X
    given parameters (a, b)
    """
    y_series = []

    for x in x_series:

        y = linear_model(x, a, b)
        y_series.append(y)

    return y_series
```

```
def linear_model(x, a, b):
    """
    linear relationship
    """
    y = a*x + b

    return y
```

HOT POTATO!

we can remove the `a, b` repeated syntax using `*args` for *clarity*

```
def predict_linear(x_series, *pars):
    """
    predicts Y for a series of X
    given parameters (a, b)
    """
    y_series = []

    for x in x_series:

        y = linear_model(x, *pars)
        y_series.append(y)

    return y_series
```

```
def linear_model(x, a, b):
    """
    linear relationship
    """
    y = a*x + b

    return y
```

UNDER THE HOOD

`*args` will take the arbitrary arguments you're passing to the function

```
my_function(3, 4, 'a')
```

and pack them into a tuple, which is then unpacked in the function

```
my_function((3, 4, 'a'))
```

**kwargs

works similarly to `*args` but uses `key:value` pairing instead

```
def database_entry(name, **ancilliary):
    """adds a new entry to a database"""
    entry = {}
    entry['name'] = name

    for (key, value) in ancilliary.items():
        entry[key] = value

    return entry
```

```
In : database_entry("Belinda")
Out: {'name': 'Belinda'}

In : database_entry("Belinda", sex="F", weight=65)
Out: {'name': 'Belinda', 'sex': 'F', 'weight': 65}
```

UNDER THE HOOD

`**kwargs` will take the arbitrary `key:value` pairs you're passing to the function

```
my_function("John", height=190, weight=80)
```

and pack them into a `dict`, and then unpacked within the function

```
my_function("John", {height=190, weight=80})
```

SUMMARY

FUNCTIONS ARE INVALUABLE

we can break down a program into much smaller parts, which can:

✔ be repeatedly executed or on demand

✔ allows testing of processes in isolation

✔ overall, simplifies a much larger problem

SOME CONVENTIONS TO REMEMBER

the name should clearly state what it does

♦

the process should be as **simple** as possible

♦

always have a *docstring* describing its purpose

♦

anyone should be able to use it

NOTE

the *docstring* may seem trivial, however its presence allows you to reference its purpose at any time like so

```
In : help(circle_area)
```

```
circle_area(radius)
    Calculates the area of a circle
```

incredibly important for → other people → future reference → your sanity

in fact if you're ever lost about what a function does

Python's `help()` function helps you get an idea

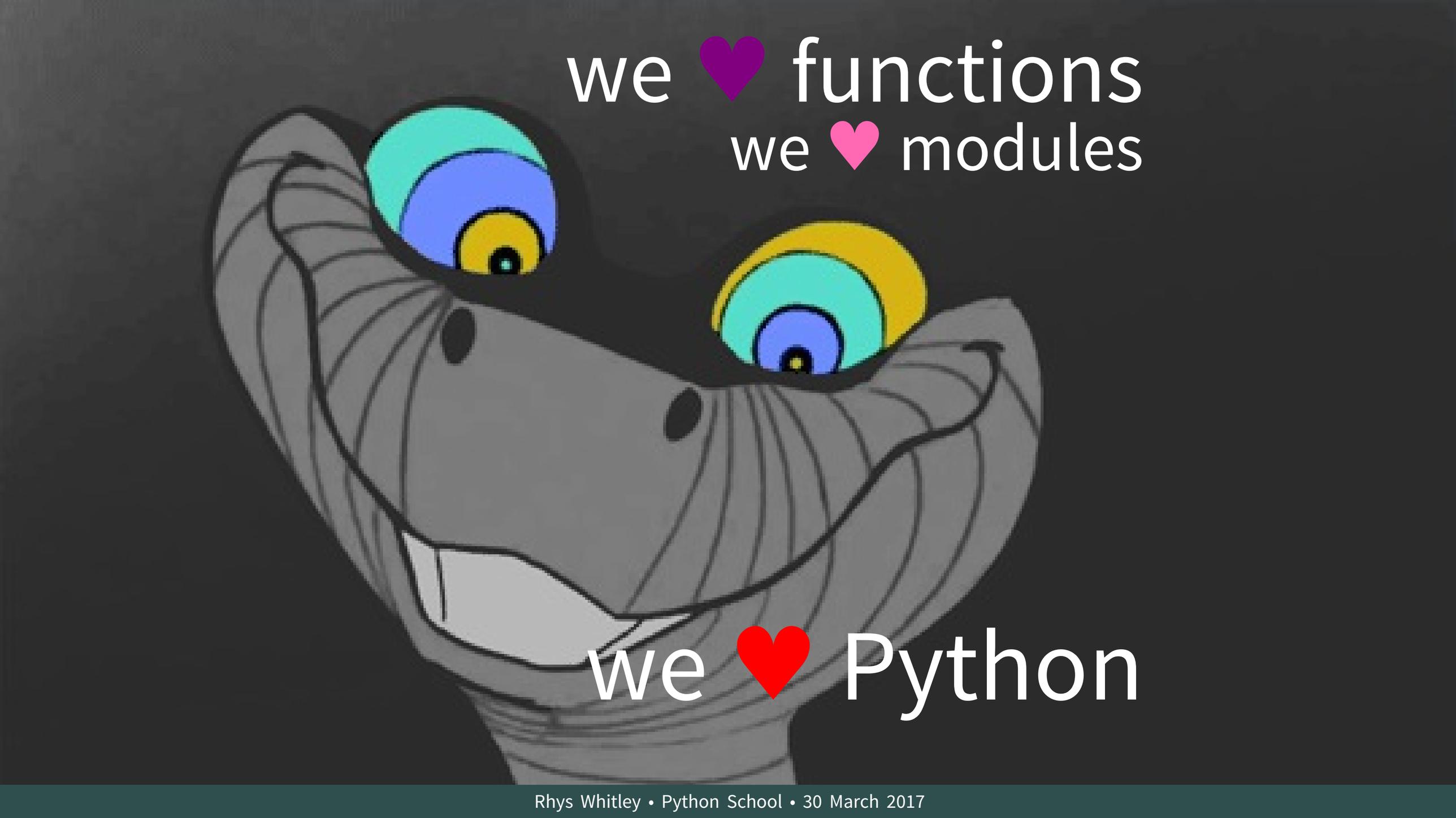
```
help(range)

          |
...verbose output...
          |
```

but when all fails seek out your best friends

Google

Stack Exchange


we ♥ functions
we ♥ modules
we ♥ Python

NEXT TIME