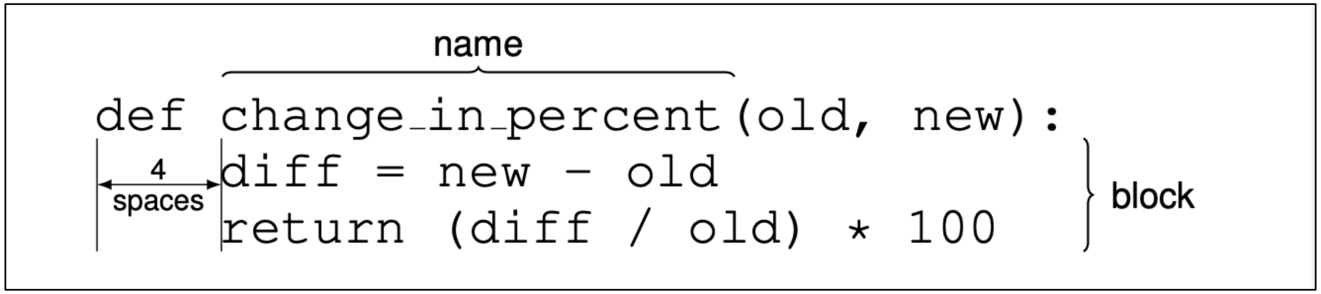
Revision: Function definition

Australian National University



- A function definition consists of a name and a body (a block)
- The extent of the block is defined by indentation, which must be the same for all statements of a block
  - Standard indentation is 4 spaces
- This example has **parameters**
  - Parameters are specified in the function call, and are passed to the code block
- A custom function must be defined before it can be called

Revision: Function parameters and `return` value

Australian National University

parameters

```
def change_in_percent(old, new):
    diff = new - old
    return (diff / old) * 100
```

- Function (formal) parameters are (variable) names
  - These variables can be used only in the function body
- Parameter values will be set only when the function is called
- `return` is a statement
  - when executed, it causes the function call to end, and return the value of the expression

Revision: Flow of execution

Australian National University

- Calling a function will interrupt the processive flow of program execution
- Calling a function causes the execution to skip to that function and continue executing from that position

```python
def ask_name():
    print("Please enter your name: ")
    name = input()
    return name

def calculate_length_of_string(the_string):
    return len(the_string)

def print_greeting(input_name):
    name_length = calculate_length_of_string(input_name)
    print("Hello, " + input_name + ".  Your name is " + str(name_length) + " characters in length.")

def interact():
    interaction_name = ask_name()
    print_greeting(interaction_name)

interact()
```

- Execution continues until the end of the function is reached (and it returns to executing where the call was originally made)

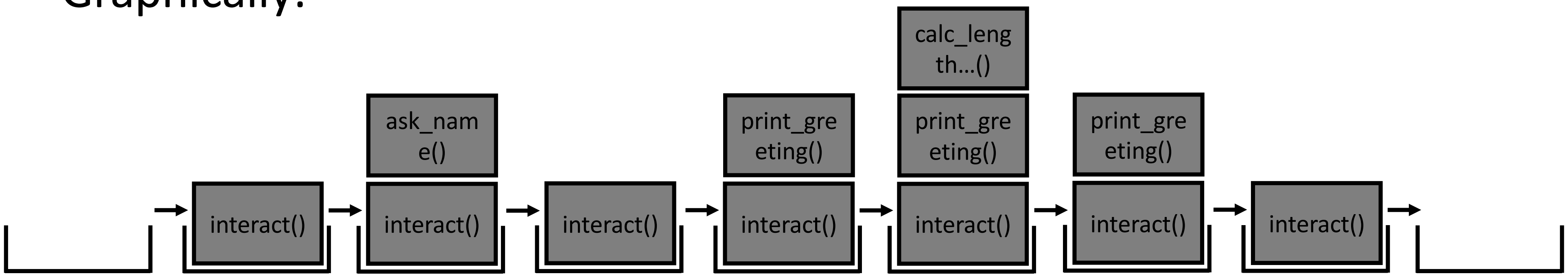
Revision: the call stack

Australian National University

- The ‘to-do list’ of where to come back to after each current function call is called the (execution or call) stack
- When evaluation of a function call begins, the current instruction sequence is put ‘on hold’ while the expression is evaluated – and the function calls begin to ‘stack up’

```
def ask_name():
    print("Please enter your name: ")
    name = input()
    return name

def calculate_length_of_string(the_string):
    return len(the_string)

def print_greeting(input_name):
    name_length = calculate_length_of_string(input_name)
    print("Hello, " + input_name + ".  Your name is " + str(name_length) + " characters in length.")

def interact():
    interaction_name = ask_name()
    print_greeting(interaction_name)

interact()
```

- Graphically:



Intro to Scope

COMP1730/3730

Chapter 3 : Sweigart, *Automate the boring stuff with Python*,

Or

Section 9: https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces

Australian National University

Reading: Scope

Australian National University

- Covered in Think Python and ItSPwP, but..
- A better introduction is in: *Automate the Boring Stuff with Python*, Chapter 3, from Section 'Local and Global Scope' until the end of the chapter
- Remember: you have access to the Safari/O'Reilly bookstore through the ANU library: https://www.oreilly.com/library-access/
  - Search for 'Automate the Boring Stuff with Python'

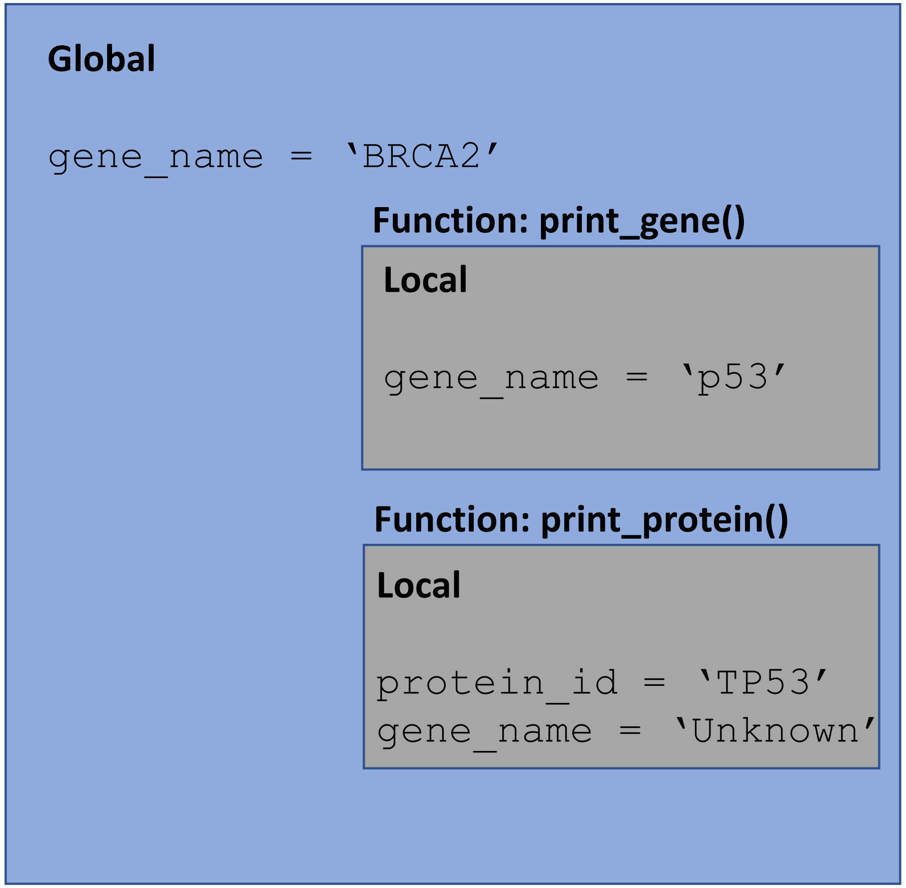
Scope - Sweigart, *Automate the Boring Stuff with Python,* Ch 3

- We haven't talked yet about scope – this is important
- So far, we have assumed that all defined variables are accessible all the time – this is known as **global** scope
- But global scope becomes hazardous as:
  - A program gets larger
  - Includes code that comes from other developers (you might both use the same variable name)
- The parameter variables within a single function are **local** to the code block. If you try to access one of these outside the function code block, you will get an error.

Graphically:

```python
def print_gene():
    gene_name = 'p53'
    print('In print_gene: ' + gene_name)

def print_protein():
    protein_id = 'TP53'
    gene_name = 'Unknown'
    print('In print_protein: ' + protein_id + ' ' + gene_name)


gene_name = 'BRCA2'
print_gene()
print('In main: ' + gene_name)
print_protein()
```

Program: scope.py



Output:

```
In print_gene: p53
In main: BRCA2
In print_protein: TP53 Unknown
```

Within a function, parameters are local

Australian National University

- Variables created/assigned in a function (including parameters) are **local** to that function:
  - Local variables have scope limited to the enclosing block
  - The interpreter uses a new namespace for each function call
  - Local variables that are not parameters are undefined before the first assignment in the function body. Then remain local to the function block
  - Variables with the same name used outside the function are unchanged after the call
- Within a function, you can still access variables in the **global** scope
- But, within function **local** scope, you *cannot* access the **local** scopes of other functions

Scope - why?

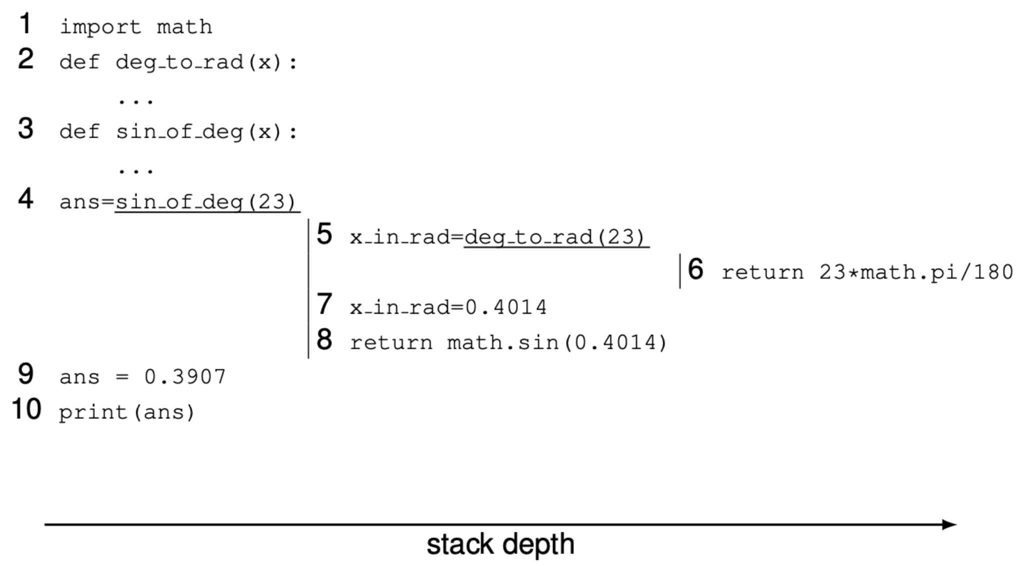
Australian National University

- There are very good reasons why every section of code should not be able to access the variables controlled by other sections.
  - For one thing, as your program gets bigger, the **namespace** of the program will start to get crowded.
  - You might be using the same variable name for two different things.
  - If you are using code from other developers (like importing functions), they might be using the same variable names as your program – but for different things
  - It makes good sense to compartmentalise variable scope, to avoid *namespace-collisions*

The call stack

Australian National University

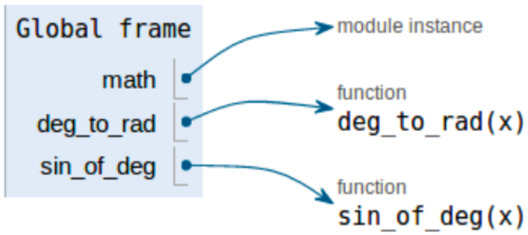
```
import math
# Convert degrees to radians
def deg_to_rad(x):
    return x * math.pi / 180

# Take sin of an angle in degrees
def sin_of_deg(x):
    x_in_rad = deg_to_rad(x)
    return math.sin(x_in_rad)

ans = sin_of_deg(23)
print(ans)
```



The call stack and scope

Australian National University

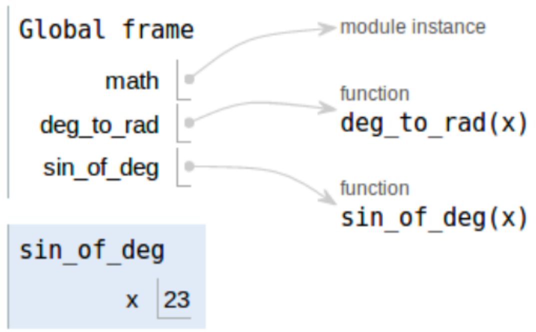
```
import math
# Convert degrees to radians
def deg_to_rad(x):
    return x * math.pi / 180

# Take sin of an angle in degrees
def sin_of_deg(x):
    x_in_rad = deg_to_rad(x)
    return math.sin(x_in_rad)

ans = sin_of_deg(23)
print(ans)
```



(Image from `pythontutor.com`)

The call stack and scope

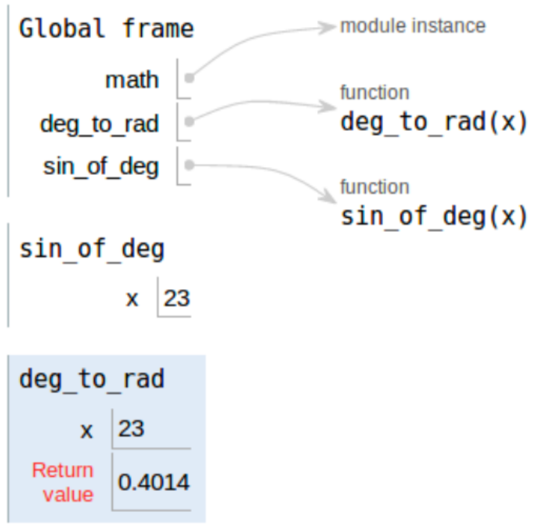
```
import math
# Convert degrees to radians
def deg_to_rad(x):
    return x * math.pi / 180

# Take sin of an angle in degrees
def sin_of_deg(x):
    x_in_rad = deg_to_rad(x)
    return math.sin(x_in_rad)

ans = sin_of_deg(23)
print(ans)
```



(Image from `pythontutor.com`)

The call stack and scope

```
import math
# Convert degrees to radians
def deg_to_rad(x):
    return x * math.pi / 180

# Take sin of an angle in degrees
def sin_of_deg(x):
    x_in_rad = deg_to_rad(x)
    return math.sin(x_in_rad)

ans = sin_of_deg(23)
print(ans)
```

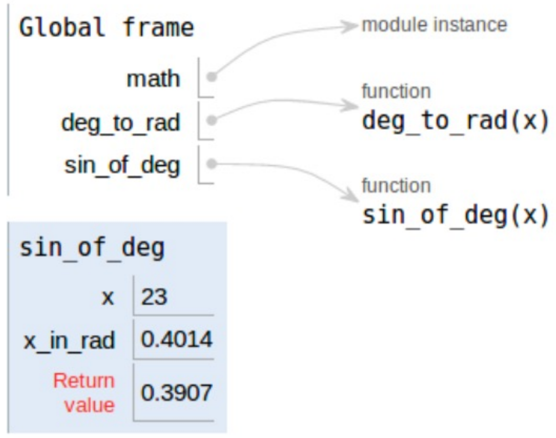


(Image from `pythontutor.com`)

The call stack and scope

Australian National University

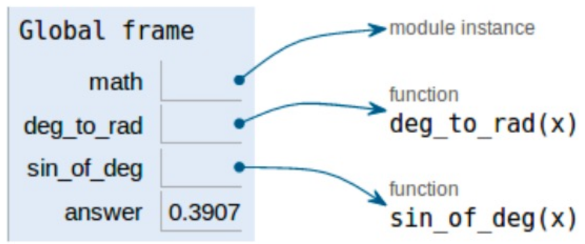
```
import math
# Convert degrees to radians
def deg_to_rad(x):
    return x * math.pi / 180

# Take sin of an angle in degrees
def sin_of_deg(x):
    x_in_rad = deg_to_rad(x)
    return math.sin(x_in_rad)

ans = sin_of_deg(23)
print(ans)
```



(Image from pythontutor.com)

The call stack and scope

```
import math
# Convert degrees to radians
def deg_to_rad(x):
    return x * math.pi / 180

# Take sin of an angle in degrees
def sin_of_deg(x):
    x_in_rad = deg_to_rad(x)
    return math.sin(x_in_rad)

ans = sin_of_deg(23)
print(ans)
```



(Image from pythontutor.com)

Functional Abstraction

COMP1730/6730

Think Python Ch 4 *(Encapsulation, Generalization, Interface Design) and* Ch 6 (*Leap of Faith*)

Australian National University

Abstraction & Interfaces

- Imagine if when we write very large programs, that we needed to understand every line of code that our code is built on?
  - It would be terrible! Nothing complicated could get done easily. Slow!
  - In the first lecture, you saw how to open a file and train a basic ML model
  - General understanding is necessary (yes!), but detailed understanding of the implementation is not
- We rely on **abstraction** of details
- We implement code and software libraries that only require an understanding of the **interface**
- When we write functions, we should make them intuitive to provide this interface

Interfaces (*Think Python*, Ch. 4)

Australian National University

- Providing a simple interface to a complex task is the great value of software libraries. Other people write code that you don't know in detail – but you can do the same tasks, with much less effort.
- Before long, you will be writing code where you didn't look at every line of the functions and libraries that you rely on.
- Ch 6 of *Think Python* calls this the 'Leap of faith':

> ## Leap of Faith
>
> Following the flow of execution is one way to read programs, but it can quickly become overwhelming. An alternative is what I call the "leap of faith". When you come to a function call, instead of following the flow of execution, you *assume* that the function works correctly and returns the right result.

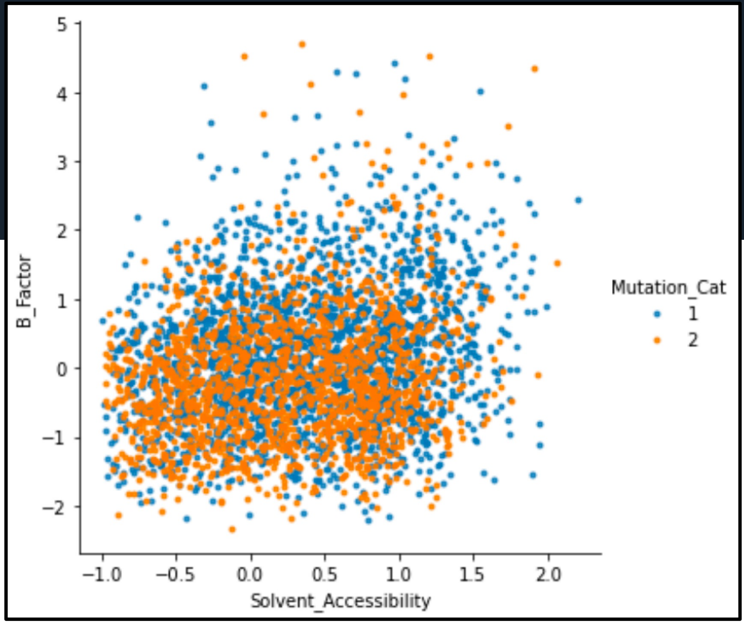
- This is abstraction. Much of the detail in your software remains abstract. You are now thinking about code at a higher level.

More complicated: Python as a toolbox

Australian National University

```python
import pandas as pd
import seaborn as sns
from sklearn.tree import DecisionTreeRegressor

mutations = pd.read_csv('/Users/dan/Desktop/Envision_manuscript_data/Gray_etal_SupplementaryTable_S2_cleaned.csv')

mutation_metrics = ["Residual_Function", "Solvent_Accessibility", "B_Factor"]

sns.lmplot( x="Solvent_Accessibility", y="B_Factor", data=mutations, fit_reg=False, hue='Mutation_Cat', legend=True, markers='.', x_jitter=True, y_jitter=True)

X = mutations.iloc[0:, [26, 30]].values # 26 == Solvent_Accessibility, 30 == B_Factor
y = mutations.iloc[0:,[6]].values # 6 is 'Mutation_Cat'

tree_clf = DecisionTreeRegressor(max_depth=2)
tree_clf.fit(X, y)

# Solvent_Accessibility: 91%
# B-factor: 187 (high, +ve)
prediction = tree_clf.predict([[2, 0.99]])
print("Residual function prediction: " + str(prediction[0]))
```

```
In [162]: runfile('/Users/dan/Desktop/second_example_v0.2.py', wdir='/Users/dan/Desktop')
Residual function prediction: 1.2415158371040724

In [163]:
```



Turtle graphics

https://docs.python.org/3/library/turtle.html

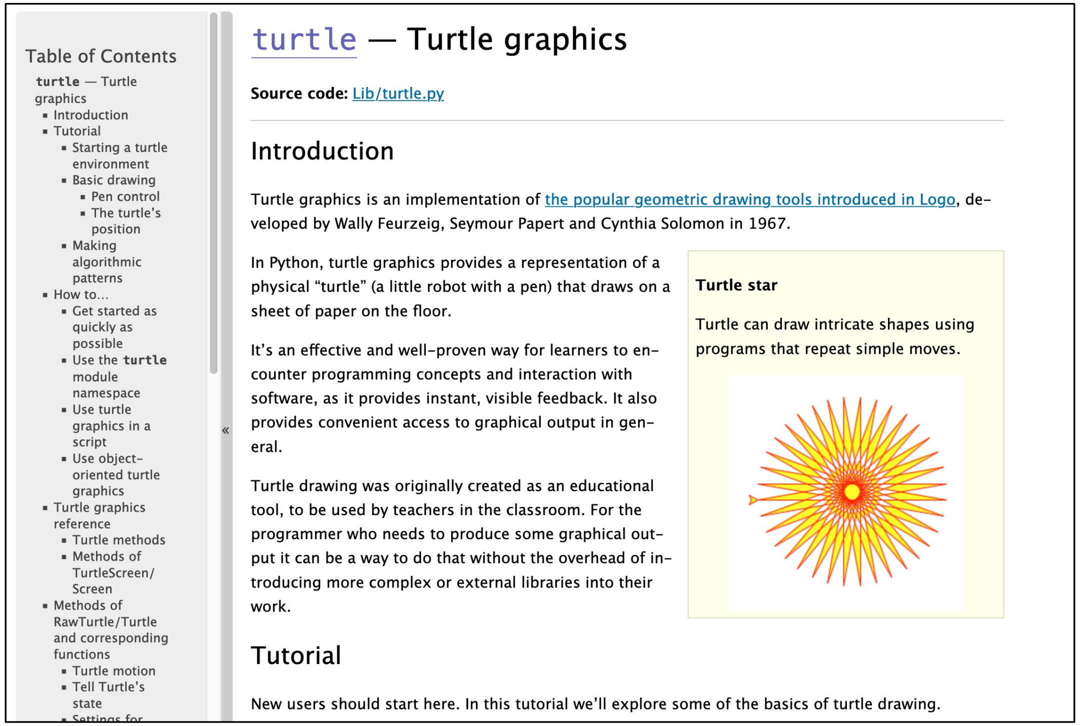


Table of Contents

turtle — Turtle graphics

Source code: Lib/turtle.py

Introduction

Turtle graphics is an implementation of the popular geometric drawing tools introduced in Logo, developed by Wally Feurzeig, Seymour Papert and Cynthia Solomon in 1967.

In Python, turtle graphics provides a representation of a physical "turtle" (a little robot with a pen) that draws on a sheet of paper on the floor.

It's an effective and well-proven way for learners to encounter programming concepts and interaction with software, as it provides instant, visible feedback. It also provides convenient access to graphical output in general.

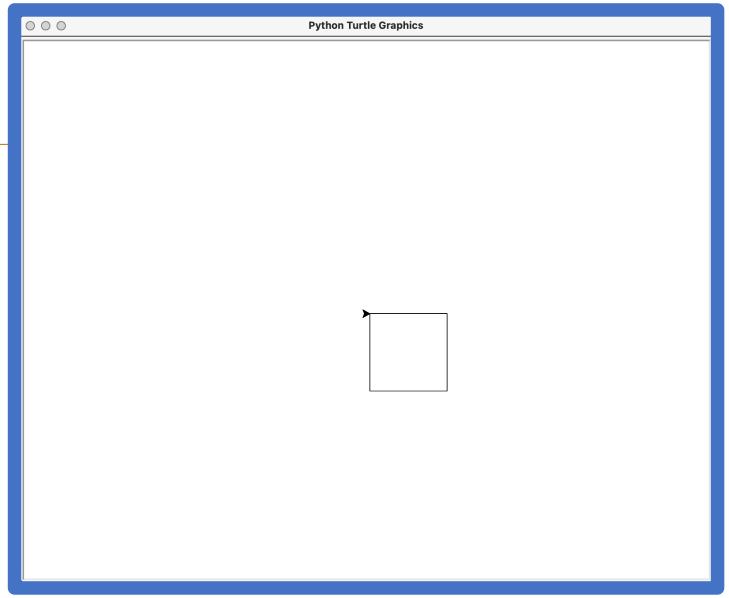
Turtle drawing was originally created as an educational tool, to be used by teachers in the classroom. For the programmer who needs to produce some graphical output it can be a way to do that without the overhead of introducing more complex or external libraries into their work.

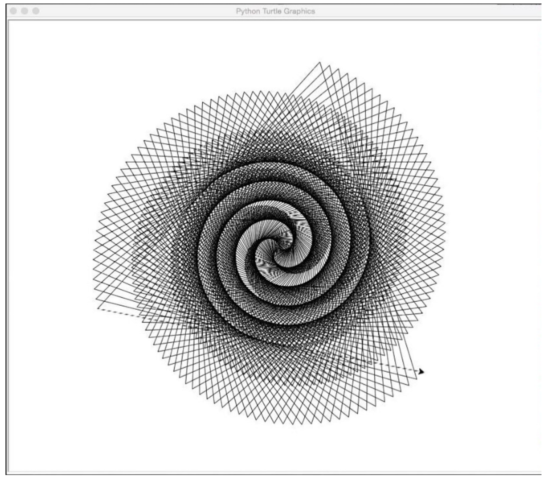
Turtle star

Turtle can draw intricate shapes using programs that repeat simple moves.

Tutorial

New users should start here. In this tutorial we'll explore some of the basics of turtle drawing.



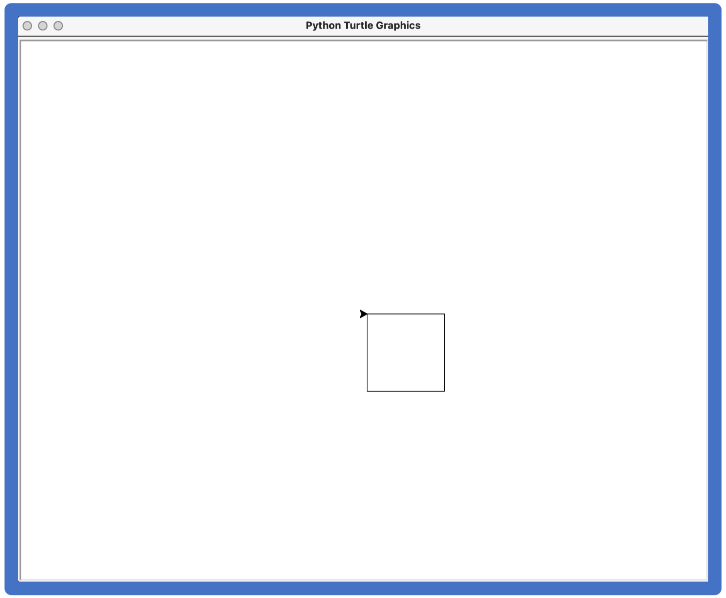


Luke Taylor, from https://www.youtube.com/watch?v=IyqTY4q16iw

Abstraction – a drawing example

Australian National University

- Let’s draw a shape - a square.




```
Console 2/A
Python 3.9.13 (main, Aug 25 2022, 18:29:29)
Type "copyright", "credits" or "license" for more
information.

IPython 8.15.0 -- An enhanced Interactive Python.

In [1]: import turtle

In [2]: turtle.forward(100)

In [3]: turtle.right(90)

In [4]: turtle.forward(100)

In [5]: turtle.right(90)

In [6]: turtle.forward(100)

In [7]: turtle.right(90)

In [8]: turtle.forward(100)

In [9]: turtle.right(90)

In [10]:
IPython Console  History
```

Code_L4_1.py

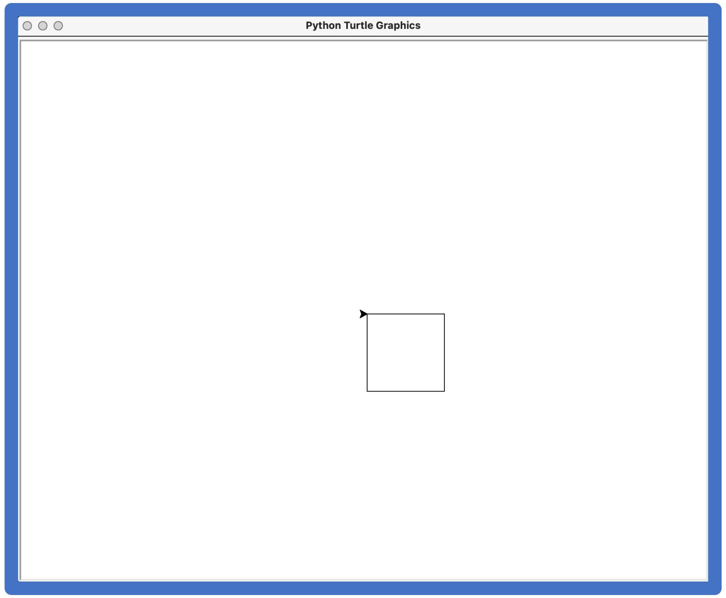
- Boring, processive, repetitive code.

Drawing shapes – a square

Australian National University

- By writing a draw_square function, we can encapsulate the complexity individual drawing commands.
- And we can add parameters

```python
import turtle

def draw_square(side_dim):
    '''Draws a square of side length defined by side_dim'''
    turtle.forward(side_dim)
    turtle.right(90)
    turtle.forward(side_dim)
    turtle.right(90)
    turtle.forward(side_dim)
    turtle.right(90)
    turtle.forward(side_dim)
    turtle.right(90)


draw_square(100)
```

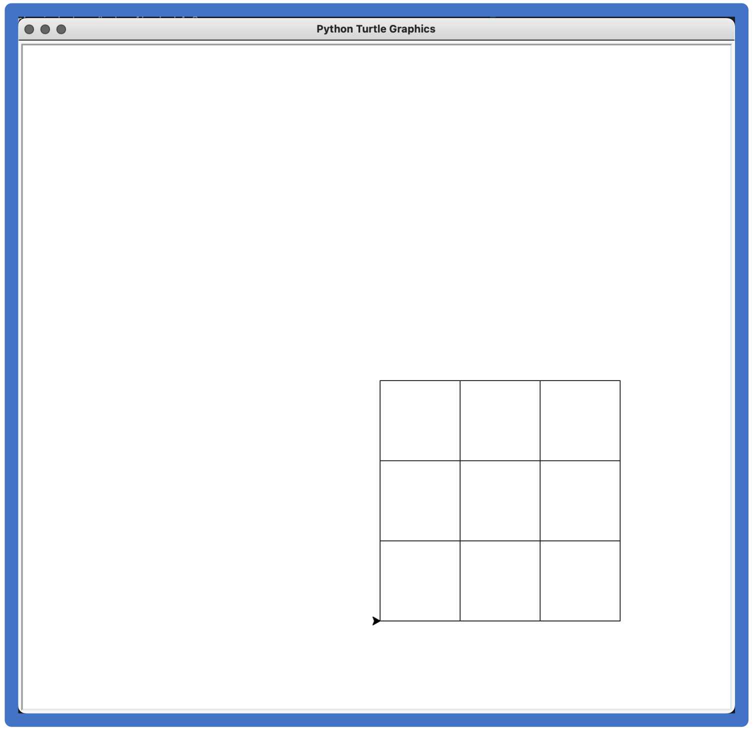


Code_L4_2.py

Drawing a bigger structure with functions

- Write another function to draw multiple squares to make a grid
- Parameters to give size and number of cells:

```
import turtle

def draw_square(side_dim):
    '''Draws a square of side length defined by side_dim'''
    turtle.forward(side_dim)
    turtle.right(90)
    turtle.forward(side_dim)
    turtle.right(90)
    turtle.forward(side_dim)
    turtle.right(90)
    turtle.forward(side_dim)
    turtle.right(90)

def draw_grid(cells, side_dim):
    '''Draws a grid of squares'''
    # draw grid
    for i in range(cells):

        # draw a row of cells
        for j in range(cells):
            draw_square(side_dim)
            turtle.forward(side_dim)

        # move to next row beginning
        turtle.penup()
        turtle.right(180)
        turtle.forward(side_dim*cells)
        turtle.left(90)
        turtle.forward(side_dim)
        turtle.left(90)
        turtle.pendown()

draw_grid(3,100)
```



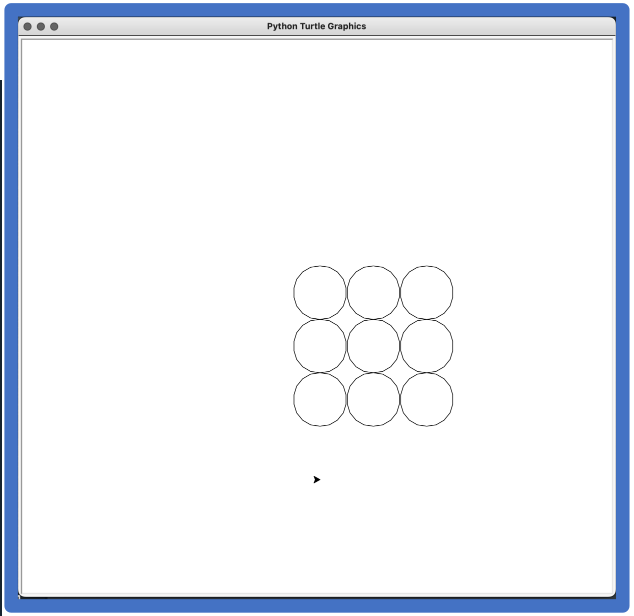
Code_L4_3.py

Abstract functions allow easy extensibility

Australian National University

- What if we now need to draw each cell as a circle – there is a function for `turtle.circle()`:

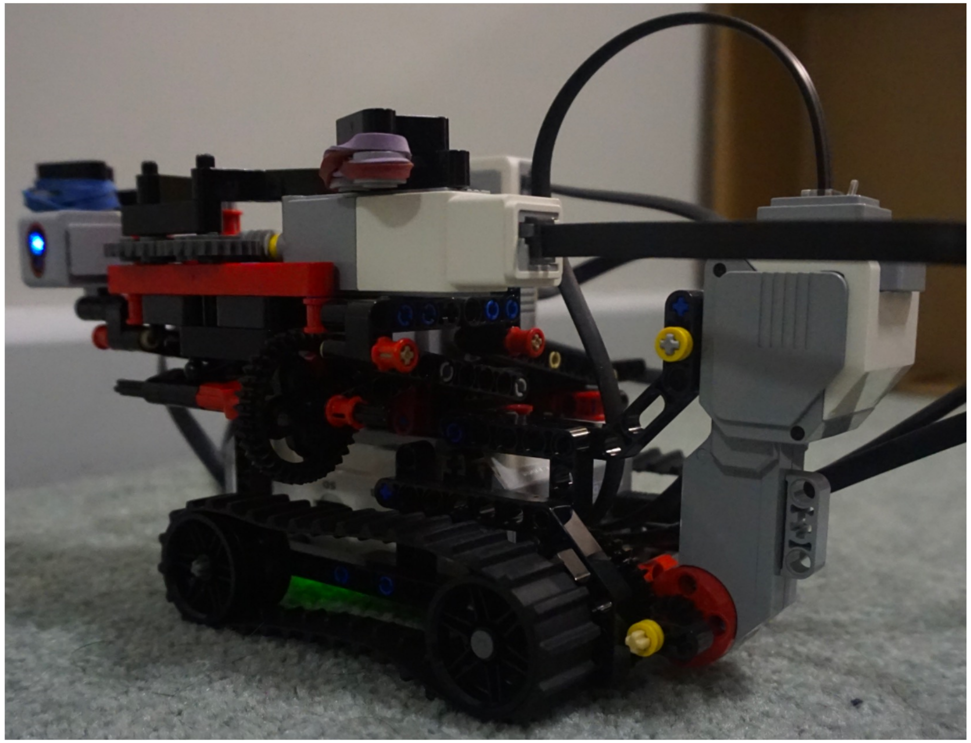
```
def draw_grid(cells, side_dim, draw_shape, adjust=1):

    # draw grid
    for i in range(cells):

        # draw a row of cells
        for j in range(cells):
            turtle.pendown()
            draw_shape(side_dim)
            turtle.penup()
            turtle.forward(side_dim*adjust)

        # move to next row beginning
        turtle.penup()
        turtle.left(180)
        turtle.forward(side_dim*cells*adjust)
        turtle.left(90)
        turtle.forward(side_dim*adjust)
        turtle.left(90)
        turtle.pendown()


#draw_grid(3, 40, draw_square, 1)
draw_grid(3, 40, turtle.circle, 2)
```

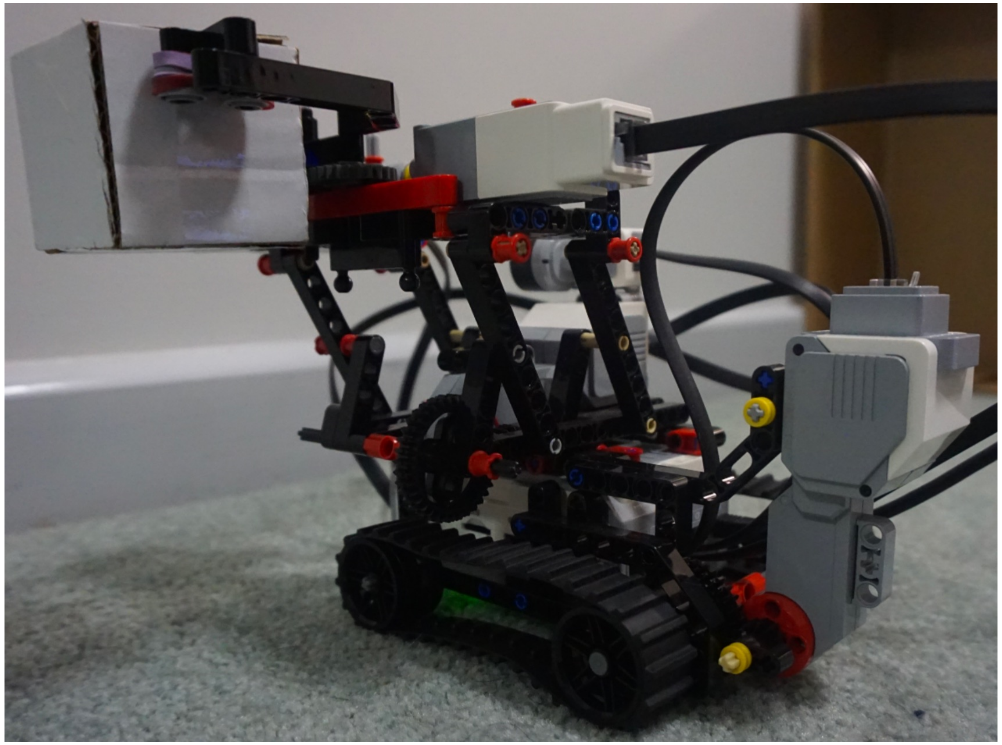


Code_L4_4.py

Example: The Robot

Australian National University

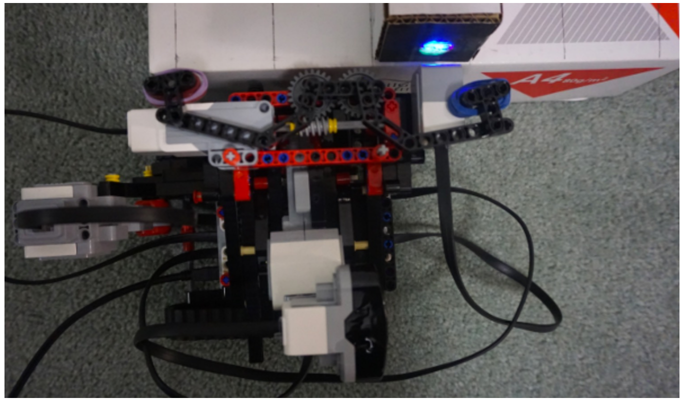


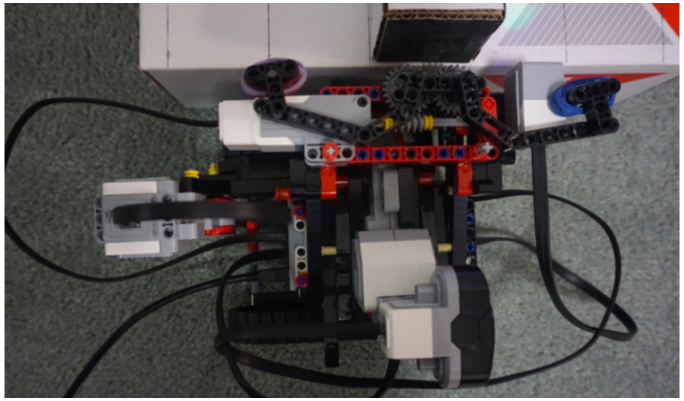


- It can:
  - Move
  - Grip boxes
  - Lift mechanical arms up and down
  - Sense position
  - Be driven by a python code library

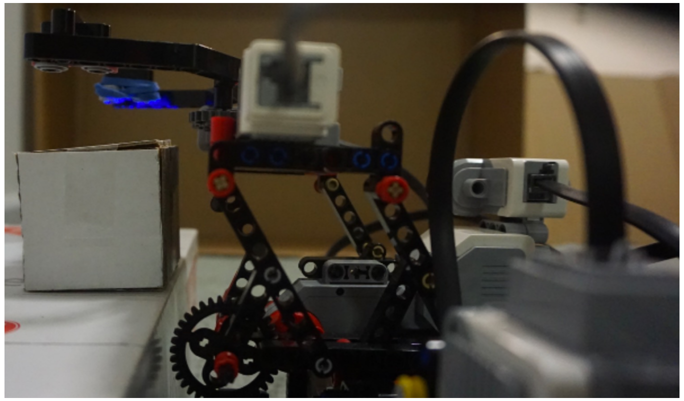
Things the robot does:

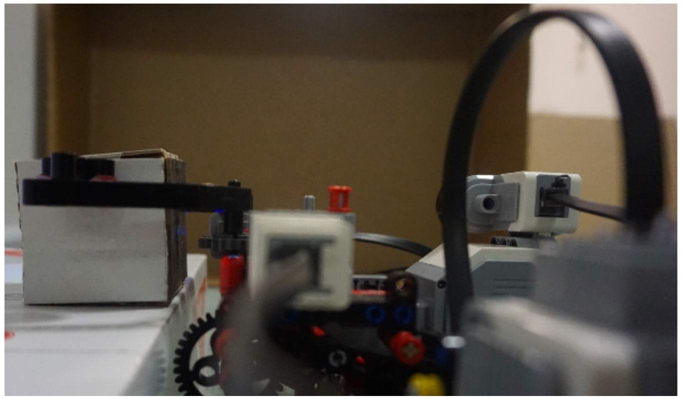
- Move left/right along a shelf with boxes on it:



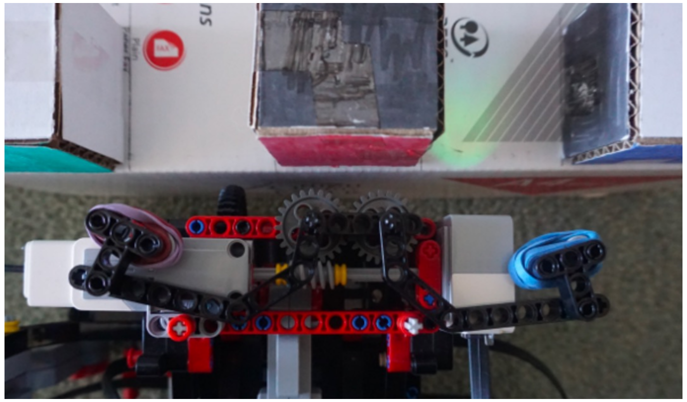


- Move gripper up and down:

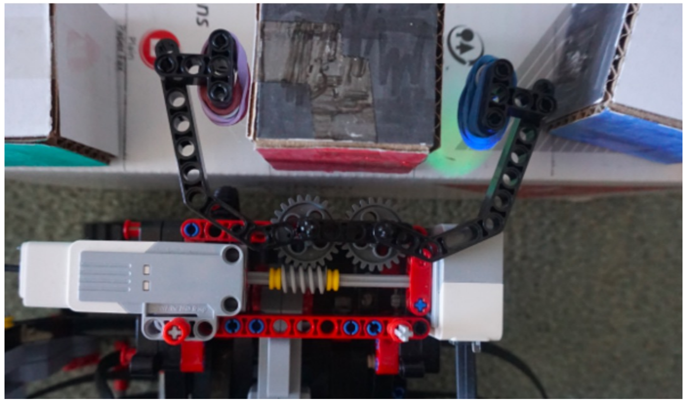




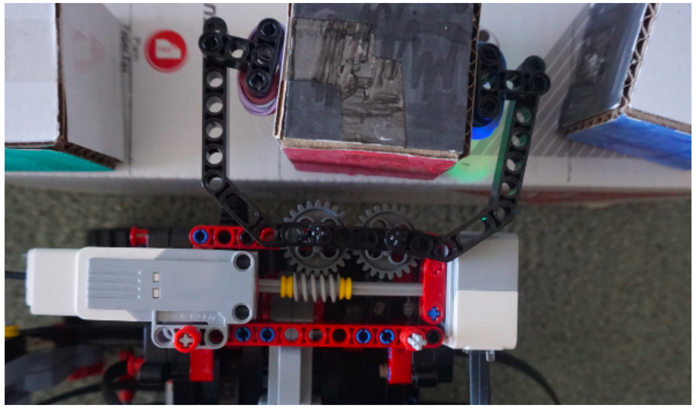
Open and close the gripper:



Folded



Open



Closed

- When moving along the shelf of boxes, the gripper needs to be folded to avoid hitting the boxes
- Folding and unfolding the gripper may hit boxes, so important to lift the gripper up first

A look at a Robot class:

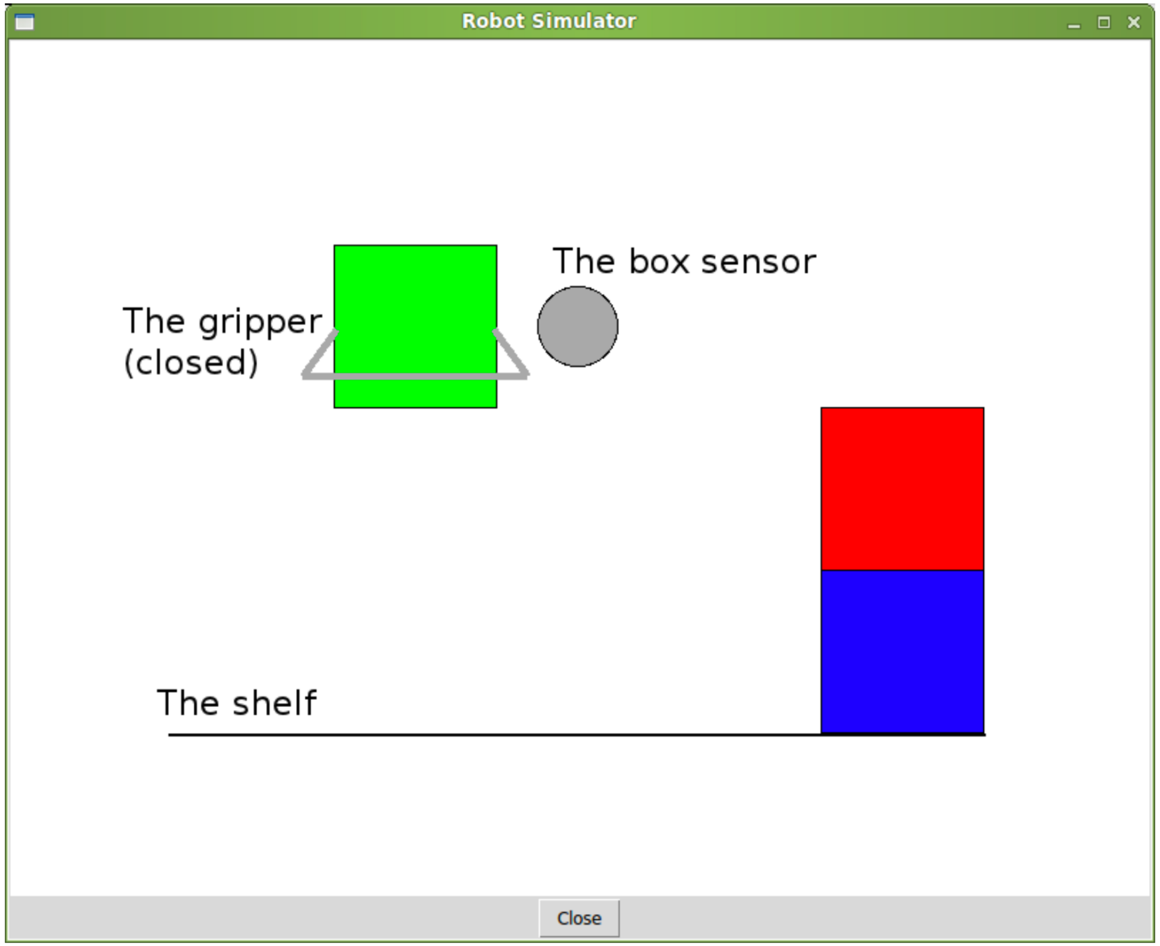
- This is the RPCRobot class that can be found in robot.py from the labs
- Class `RPCRobot`
- Global attribute defaults
- `__init__` method
- Methods:
  - `lift_up`
  - `lift_down`
  - `drive_right`
  - Etc…

```
class RPCRobot:
    '''Robot class interfacing with ev3 via RPYC.'''
    DEFAULT_DRIVE_RIGHT = 575
    DEFAULT_DRIVE_LEFT = 600
    DEFAULT_LIFT_UP = 220
    DEFAULT_LIFT_DOWN = 200

    def __init__(self, ip_address = "192.168.0.1"):
        import rpyc

        self.rpcconn = rpyc.classic.connect(ip_address)
        self.ev3 = self.rpcconn.modules.ev3dev.ev3
        self.battery = self.ev3.PowerSupply()
        self.drive = self.ev3.LargeMotor('outB')
        self.lift = self.ev3.LargeMotor('outD')
        self.gripper = self.ev3.MediumMotor('outC')
        self.sensor = self.ev3.ColorSensor()
        self.proxor = self.ev3.InfraredSensor()

    def print_state(self):
        print("drive at " + str(self.drive.position))
        print("lift at " + str(self.lift.position))
        print("gripper at " + str(self.gripper.position))
        print("sensor read: " + str(self.sensor.value()))
        print("proxor read: " + str(self.proxor.value()))
        print("battery: " + str(self.battery.measured_volts) + "V, "
              + str(self.battery.measured_amps) + "A")

    # moving up doesn't require braking
    def lift_up(self, distance=DEFAULT_LIFT_UP):
        print("lift at " + str(self.lift.position)
              + ", speed " + str(self.lift.speed))
        self.lift.run_to_rel_pos(position_sp = -distance, duty_cycle_sp = -25)
        time.sleep(0.5)
        while abs(self.lift.speed) > 0:
            print("lift at " + str(self.lift.position)
                  + ", speed " + str(self.lift.speed))
            time.sleep(0.25)
        print("(end) lift at " + str(self.lift.position)
              + ", speed " + str(self.lift.speed))

    # moving down requires braking, and even then has to be commanded ~10 short
    def lift_down(self, distance=DEFAULT_LIFT_DOWN):
        print("lift at " + str(self.lift.position)
              + ", speed " + str(self.lift.speed))
        self.lift.run_to_rel_pos(position_sp = distance,
                                 duty_cycle_sp = 25,
                                 stop_command='brake')
        time.sleep(0.5)
        while abs(self.lift.speed) > 0:
            print("lift at " + str(self.lift.position)
                  + ", speed " + str(self.lift.speed))
            time.sleep(0.25)
        print("(end) lift at " + str(self.lift.position)
              + ", speed " + str(self.lift.speed))

    def drive_right(self, distance = DEFAULT_DRIVE_RIGHT):
        print("drive at " + str(self.drive.position)
              + ", speed " + str(self.drive.speed))
        self.drive.run_to_rel_pos(position_sp = distance,
                                  duty_cycle_sp = 50,
                                  stop_command='hold')
        time.sleep(0.5)
```

Australian National University

The Robot Simulator

Australian National University



The `robot` library

```
>>> import robot
```

Start new simulation:

```
>>> robot.init()
```

Start simulation with larger area:

```
>>> robot.init(width = 11, height = 6)
```

Start simulation with random boxes:

```
>>> robot.init(width = 11, height = 6,
               boxes = "random")
```

Drive right/left one step:

```
>>> robot.drive_right()
>>> robot.drive_left()
```

The robot library

Move the lift up one step:

```
>>> robot.lift_up()
```

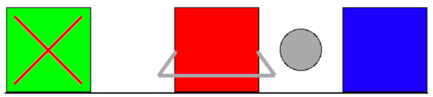
Move the lift down one step:

```
>>> robot.lift_down()
```

Change gripper position:

```
>>> robot.gripper_to_open()
>>> robot.gripper_to_closed()
>>> robot.gripper_to_folded()
```

* If the robot hits a box, no command works until a new simulation is started.



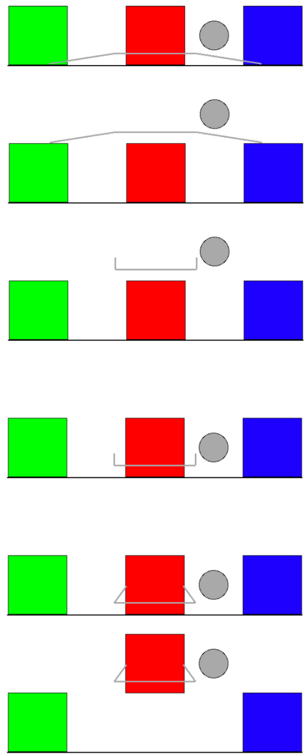
How to pick up a box?

Australian National University

- How to pick up a box without hitting the box(es) next to it?

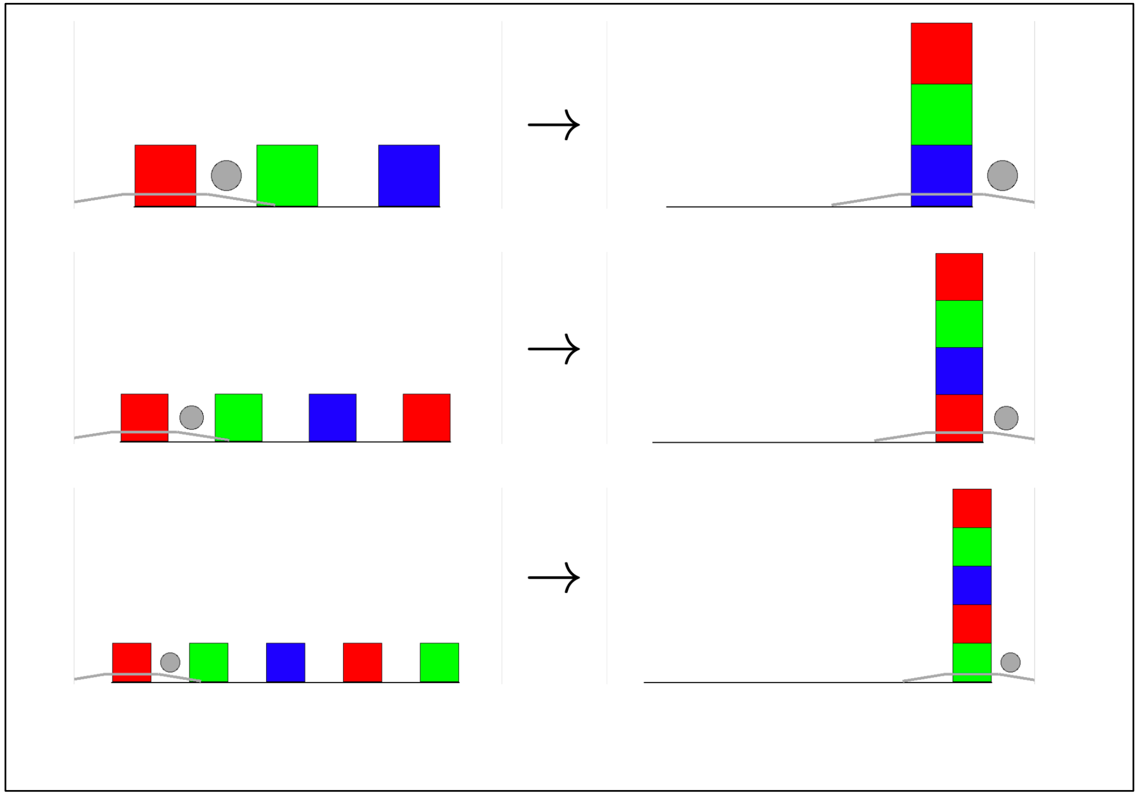
```
robot.lift_up()
robot.gripper_to_open()
robot.lift_down()
robot.gripper_to_closed()
robot.lift_up()
```

- A *program* is a sequence of instructions.



How to build a tower of boxes?

Australian National University



```
robot.init(width = 7, boxes = "flat")
robot.drive_right()
robot.lift_up()
robot.gripper_to_open()
robot.lift_down()
robot.gripper_to_closed()
robot.lift_up()
robot.drive_right()
robot.drive_right()
robot.gripper_to_open()
robot.lift_down()
robot.gripper_to_closed()
robot.lift_up()
robot.drive_right()
robot.drive_right()
robot.gripper_to_open()
robot.lift_down()
⋮
```

- This quickly gets very tedious!

Abstraction with functions

- We only need to know what a function does. We don't need to know how it does it:

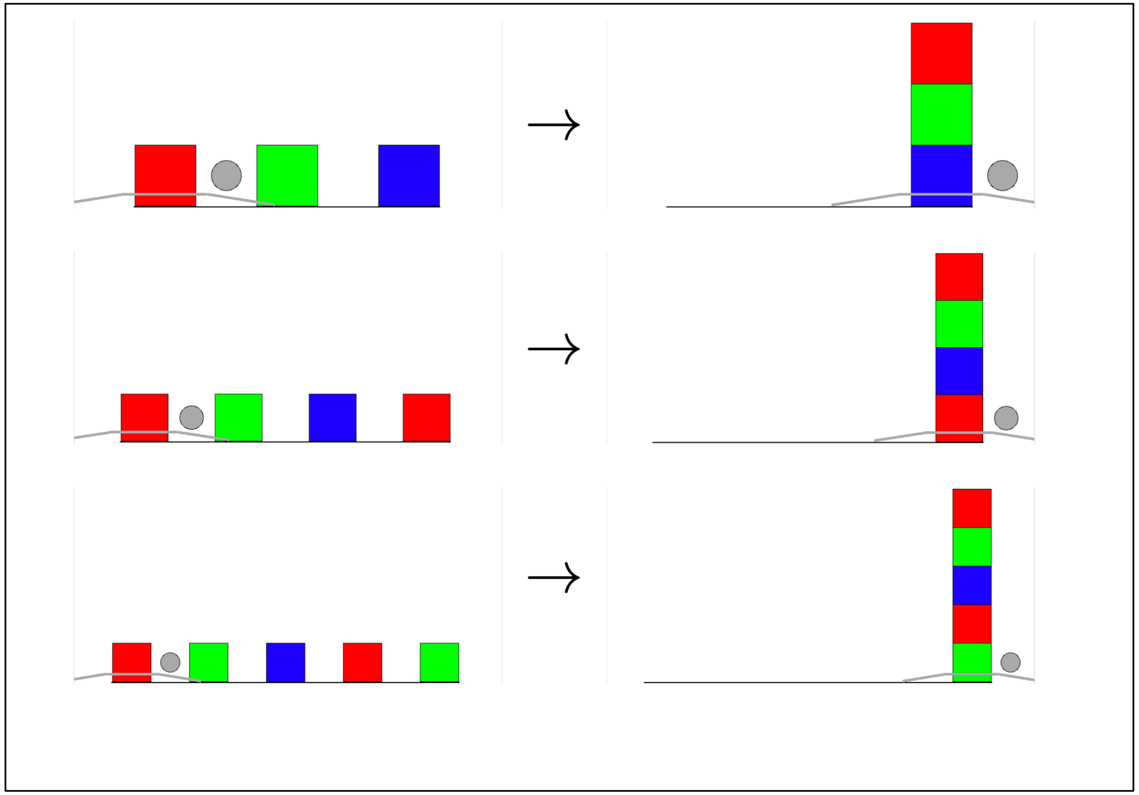
```
def grasp_box_on_shelf():
    robot.lift_up()
    robot.gripper_to_open()
    robot.lift_down()
    robot.gripper_to_closed()
    robot.lift_up()
```

4 spaces

- And the idea is, there is a high-level function to do all the necessary tasks:

```
def release_and_pickup_next():
    robot.gripper_to_open()
    robot.lift_down()
    robot.gripper_to_closed()
    robot.lift_up()
```

How to build a tower of boxes?



```
robot.init(width = 9, boxes = "flat")
robot.drive_right()
grasp_box_on_shelf()
move_to_next_stack()
release_and_pickup_next()
move_to_next_stack()
release_and_pickup_next()
move_to_next_stack()
release_and_pickup_next()
move_to_next_stack()
robot.gripper_to_folded()
robot.lift_down()
```

- Much better!
- And you needn't stop there:

`build_tower()` ?

Exercises

- Exercises in this week's practical lab

Reading

Think Python Ch 4 *(Encapsulation, Generalization, Interface Design) and* Ch 6 (*Leap of Faith*)