

# COMP1730/COMP6730

## Programming for Scientists

### Control, part 1: Branching



# Outline

- \* Program control flow
- \* Branching: The `if` statement
- \* Recursion



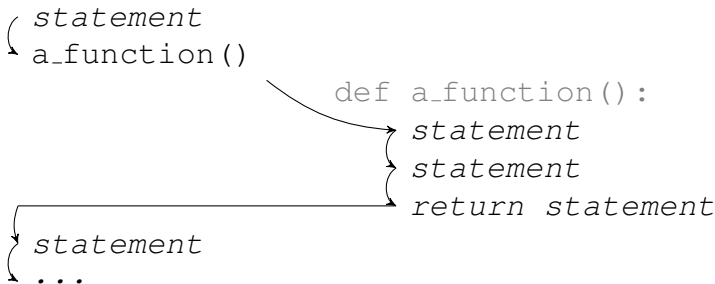
# Program control flow

# Sequential program execution

*statement*  
}  
*statement*  
}  
*statement*  
}  
*statement*  
}  
...

- \* The python interpreter always executes instructions (statements) one at a time in sequence.

*statement*  
a\_function()  
  
def a\_function():  
    *statement*  
    *statement*  
    return *statement*  
  
*statement*  
...



- ★ Function calls “insert” a function suite into this sequence, but the sequence of instructions remains invariably the same.

# Branching program flow

```
if test:  
    statement  
    statement  
    ...  
else:  
    statement  
    statement  
    ...  
statement  
...
```

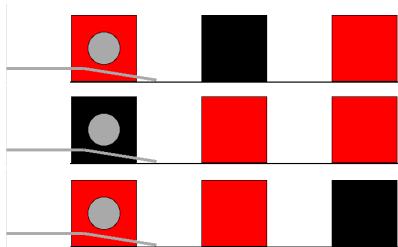
OR

```
if test:  
    statement  
    statement  
    ...  
else:  
    statement  
    statement  
    ...  
statement  
...
```

- \* Depending on the outcome of a test, the program executes one of two alternative branches.

# Problem: Stack the red boxes

- \* Two of three boxes on the shelf are red, and one is not; stack the two red boxes together.
- \* Write a program that works wherever the red boxes are.



- \* `robot.sense_color()` returns the color of the box in front of the sensor, or no color ( `' '` ) if no box detected.



```
>>> robot.sense_color()  
'red'
```

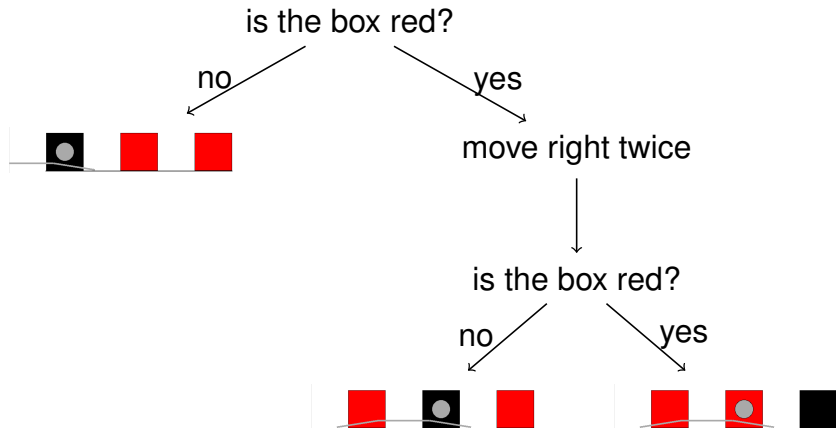


```
>>> robot.sense_color()  
' '
```

- Note that the color name is a string (in `' '`)
- The box sensor is one step right of the gripper.



# Algorithm idea



# The `if` statement

```
if test_expression :  
    suite  
statement (s)
```

1. Evaluate the test expression (converting the value to type `bool` if necessary).
2. If the value is `True`, execute the suite, then continue with the following statements (if any).
2. If the value is `False`, skip the suite and go straight to the following statements (if any).

# The `if` statement, with `else`

```
if test_expression :  
    suite_1  
else:  
    suite_2  
statement (s)
```

1. Evaluate the test expression.
2. If the value is `True`, execute suite #1, then following statements (if any).
2. If the value is `False`, execute suite #2, then following statements (if any).

# Truth values (reminder)

- \* Type `bool` has two values: `False` and `True`.
- \* Boolean values are returned by comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) and a few more.
- \* Ordering comparisons can be applied to pairs of values of the same type, for (almost) any type.
- \* *Warning #1*: Where a truth value is required, python automatically converts any value to type `bool`, but it may not be what you expected.
- \* *Warning #2*: Don't use arithmetic operators (`+`, `-`, `*`, etc.) on truth values.

# Suites (reminder)

- \* A *suite* is a (sub-)sequence of statements.
- \* A suite must contain at least one statement!
- \* In python, a suite is delimited by indentation.
  - All statements in the suite **must be preceded by the same number of spaces/tabs** (standard is 4 spaces).
  - The indentation depth of the suite inside an `if` (and `else`) statement must be greater than that of the `if` (`else`).
- \* A suite can include nested suites (`if`'s, etc).

# Suites: A side remark

- \* (Almost) Every programming language has a way of grouping statements into suites/blocks.
  - For example, in C, Java and many other:

```
if (expression) {  
    suite  
}
```

- or in Ada or Fortran (post -77):

```
if expression then  
    suite  
end if
```

- \* The use of indentation to *define* suites is a python peculiarity.



```
def stack_red_boxes():
    if robot.sense_color() == 'red':
        robot.drive_right()
        robot.drive_right()
        if robot.sense_color() == 'red':
            # stack middle box on left
        else:
            # stack left box on right
    else:
        # stack middle box on right
```



```
def print_grade(mark):  
    if mark >= 80:  
        print("HD")  
    if mark >= 70:  
        print("D")  
    if mark >= 60:  
        print("Cr")  
    if mark >= 50:  
        print("P")  
    if mark < 50:  
        print("Fail")
```

\* What will `print_grade(55)` print?



# Boolean operators

- \* The operators `and`, `or`, and `not` combine truth values:

$a$ and $b$	True iff $a$ and $b$ both evaluate to True.
$a$ or $b$	True iff at least one of $a$ and $b$ evaluates to True.
not $a$	True iff $a$ evaluates to False.

- \* Boolean operators have lower precedence than comparison operators (which have lower precedence than arithmetic operators).



```
def print_grade(mark):  
    if mark >= 80:  
        print("HD")  
    if mark < 80 and mark >= 70:  
        print("D")  
    if mark < 70 and mark >= 60:  
        print("Cr")  
    if mark < 60 and mark >= 50:  
        print("P")  
    if mark < 50:  
        print("Fail")
```



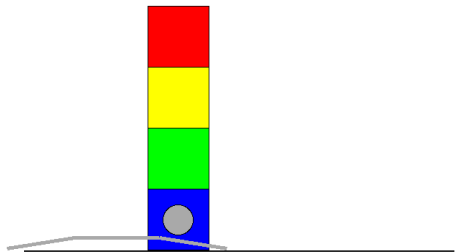
# Recursion

# Recursion

- ★ The suite of a function can contain function calls, including *calls to the same function*.
  - This is known as *recursion*.
- ★ The function suite must have a branching statement, such that a recursive call does not always take place (“base case”); otherwise, recursion never ends.
- ★ Recursion is a way to think about solving a problem: how to reduce it to a simpler instance of itself?

# Problem: Counting boxes

- \* How many boxes are in the stack from the box in front of the sensor and up?



- \* If `robot.sense_color() == ''`, then the answer is zero.
- \* Else, one plus what the answer would be if the lift was one level up.



```
def count_boxes():  
    if robot.sense_color() == '':  
        return 0  
    else:  
        robot.lift_up()  
        num_above = count_boxes()  
        robot.lift_down()  
        return 1 + num_above
```

# The call stack (reminder)

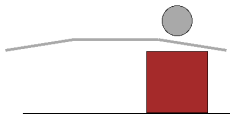
- \* When a function call begins, the current instruction sequence is put “on hold” while the function suite is executed.
- \* Execution of a function suite ends when it encounters a `return` statement, or reaches the end of the suite.
- \* The interpreter then returns to the next instruction after where the function was called.
- \* The *call stack* keeps track of where to come back to after each current function call.



```
1 ans = count_boxes()
```

```
2 if robot.sense_color() == '':
```

```
3 robot.lift_up()
```



```
4 num_above = count_boxes()
```

```
5 if robot.sense_color() == '':
```

```
6 return 0
```

```
7 num_above = 0
```

```
8 robot.lift_down()
```



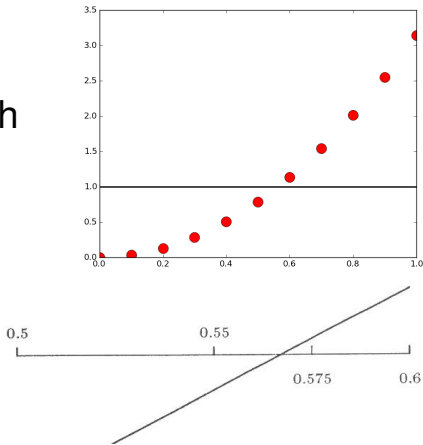
```
9 return num_above + 1
```

```
10 ans = 1
```



# Problem: Solving an equation

- \* Solve  $f(x) = 0$ .
- \* For example, find  $r$  such that  $r^2\pi = 1$ .
- \* The interval-halving algorithm.



- \* Assumption:  $f(x)$  is monotone increasing and crosses 0 in the interval  $[l, u]$ .
- \* Idea:
  - Find the middle of the interval,  $m$ :
  - if  $f(m) \approx 0$ , we're done;
  - if  $f(m) < 0$ , the solution lies in  $[m, u]$ ;
  - if  $f(m) > 0$ , the solution lies in  $[l, m]$ .

- \* *Never compare floats with `==`.*

