# COMP1730/COMP6730
Programming for Scientists

Control, part 1: Branching



## Announcements

*Also check news and announcement forum on Wattle*

* Fill out week 2-3 survey on Wattle.
  - We may make adjustment(s) to improve your learning experience.
* Homework 2 due by Sunday, 13/8/2023, 11:55pm.
* Week 3 quiz on Wattle and Lab 2 are also released.
* Four class representatives chosen.

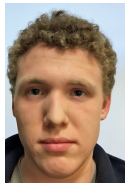


## Class reps

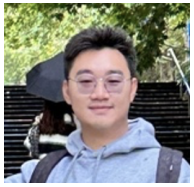
Aishwarya Sonavane

Doris Ding

Jono Granger



Vision Zhang



`https://comp.anu.edu.au/courses/comp1730/communication/`

*Give them feedback if you don't want to tell us directly (see Wattle for their emails).*



## 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 expression
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.

## The `if` statement

```
if test_expression:
    suite

other_statements()
```

*Statements within the suite must have equal indentation.*

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).
3. If the value is `False`, skip the suite and go straight to the following statements (if any).

Australian National University

## Example: Absolute difference

```
def absolute_difference(num1, num2):
    diff = num1 - num2
    if diff < 0:
        diff = diff * -1
    return diff

adiff = absolute_difference(-5, 3)
print("absolute difference is", adiff)
```

Australian National University

## The `if` statement, with `else`

```
if test_expression:
    suite_1
else:
    suite_2

other_statements()
```

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

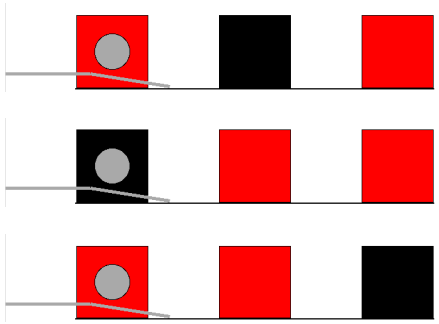
Australian National University

## Example: Absolute difference

```
def absolute_difference(num1, num2):
    if num1 >= num2:
        return num1 - num2
    else:
        return num2 - num1

adiff = absolute_difference(-5, 3)
print("absolute difference is", adiff)
```

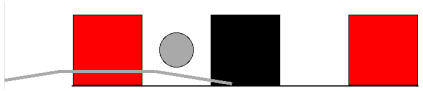
## Programming problem: Stack the red boxes

- 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'
```

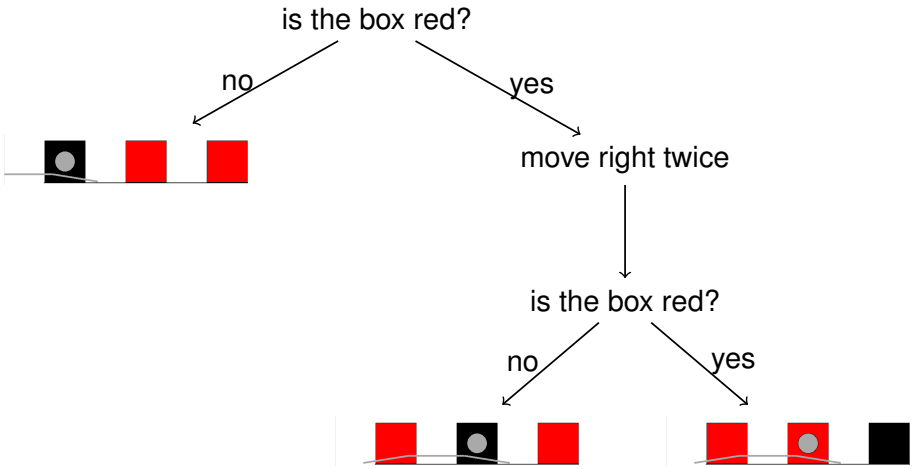


```
>>> robot.sense_color()
''
```

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

Australian National University

## Algorithm idea




Australian National University

## 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: 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 print_grade(mark):
    """Print corresponding grade for the 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")
```

* Is this code correct?

Australian National University

## 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):
    """Print corresponding grade for the 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")
```

## The `if–elif–else` statement

```
if bool_exp_1:
    suite_1
elif bool_exp_2:
    suite_2
elif bool_exp_3:
    suite_3
...
else:
    else_suite

statement(s)
```

- Tests are evaluated in sequence, and only the suite corresponding to the first test that returns `True` is executed.
- The `else` suite is executed only if all tests return `False`.

```
def print_grade(mark):
    """Print corresponding grade for the mark"""
    if mark >= 80:
        print("HD")
    elif mark >= 70:
        print("D")
    elif mark >= 60:
        print("Cr")
    elif mark >= 50:
        print("P")
    else:
        print("Fail")
```

Australian National University

Recursion

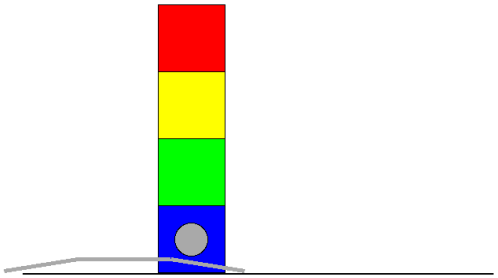
Australian National University

## 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.

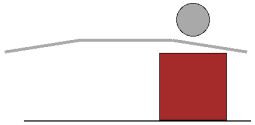
```
def count_boxes():
    if robot.sense_color() == '':
        return 0
    else:
        robot.lift_up()
        num_above = count_boxes()

        # also works without lift_down, added to move robot back
        # to the original position
        robot.lift_down()

        return 1 + num_above
```

Australian National University

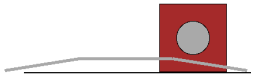
# The call stack (reminder)

* When a function call begins, the current instruction of the caller function is put “on a stack”.
* The called function 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
```

# Take home message

* Branching (`if`) statement allows a program to alter the sequence of the statements depending on some condition.
* Recursion is used to solve the current problem by looking at a simpler version of the same problem.
* Recursive calls must occur in a branching statement so that it does not run forever.