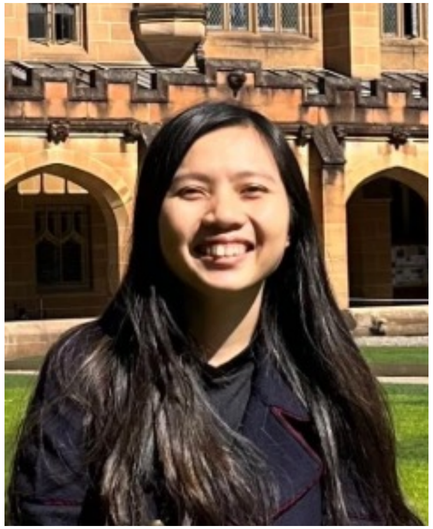
Announcements

- Please fill out the Week 2/3 Course Survey on Wattle
  - Survey comments allow us to actively adjust the course as it is taught
- Homework 2 has been released and it is due next Sunday night (04/03/24)
- Quiz for Week 3 also released
- Class representatives have been chosen

Course representatives

- COMP1730:
  - Clarissa Blum
  - Conor Aloisi
- COMP6730:
  - Thi Do
  - Xi (Darcy) Ding



Thi Do

Xi (Darcy) Ding

- Contact details are posted on Wattle site

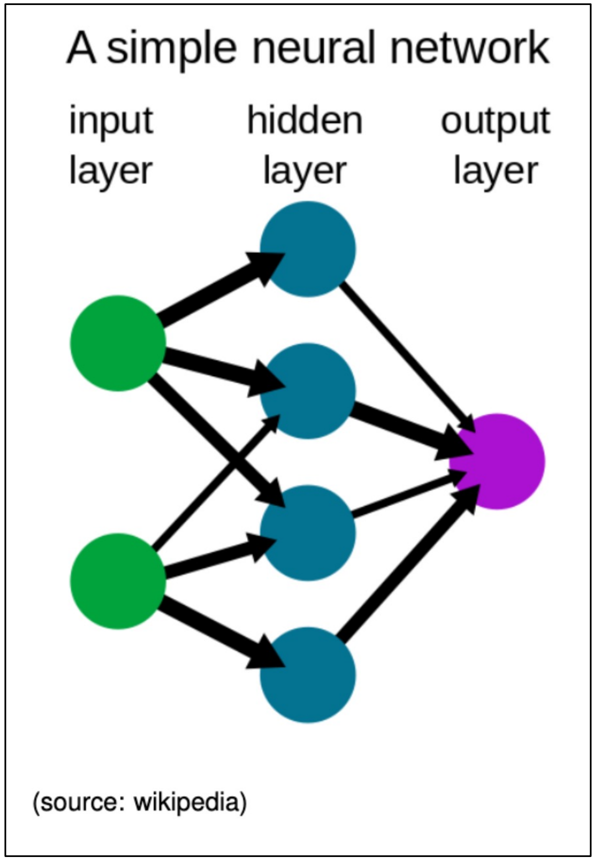
Lecture Roadmap

Australian National University

- Intro to Programming
- Variables
- Functions
  - Definitions
  - The stack
  - Scope
  - Functional abstraction
- Flow control – branching, recursion and iteration
  - branching
  - recursion
  - iteration
- Strings
- Lists
- Code quality
- File IO
- Modules & Classes

Example: Neural networks

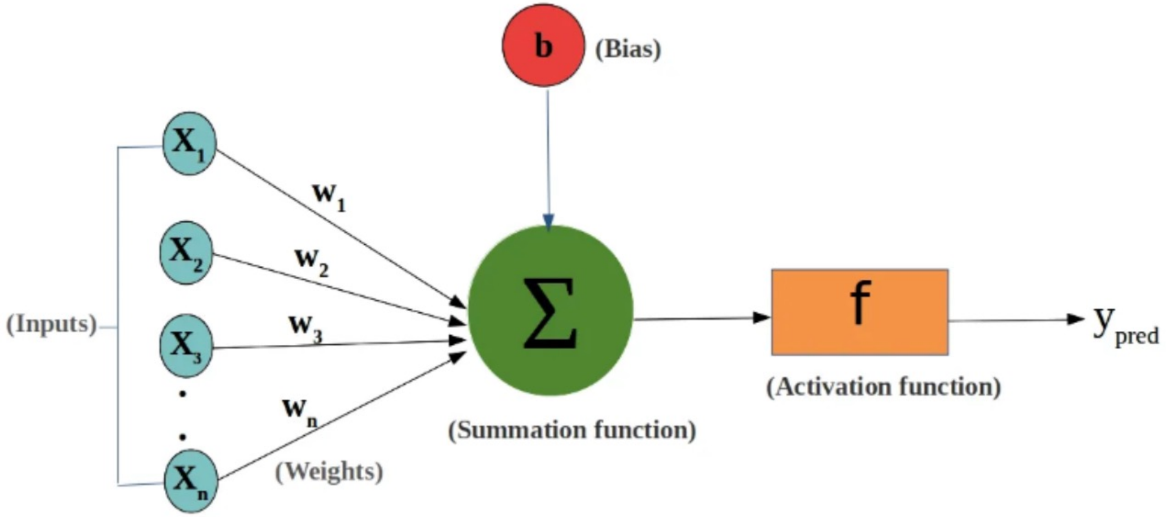
Australian National University



(source: wikipedia)

- Neural networks are mathematical representations that learn the relationship between input and output values
- Each node represents an artificial neuron
- The arrows represent connections between the outputs of one node and the input to another
- The connections have different weights – represented by thickness of the arrows
- The inputs and the weights across the network can be used to calculate the output value

Calculation of the output from a single neuron

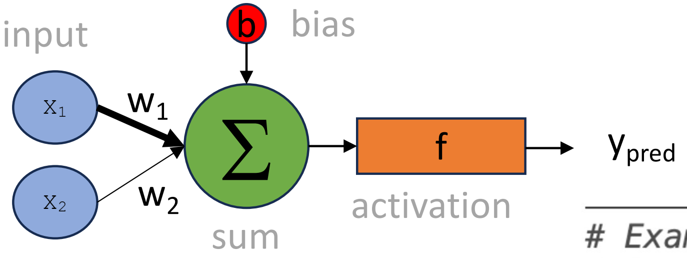


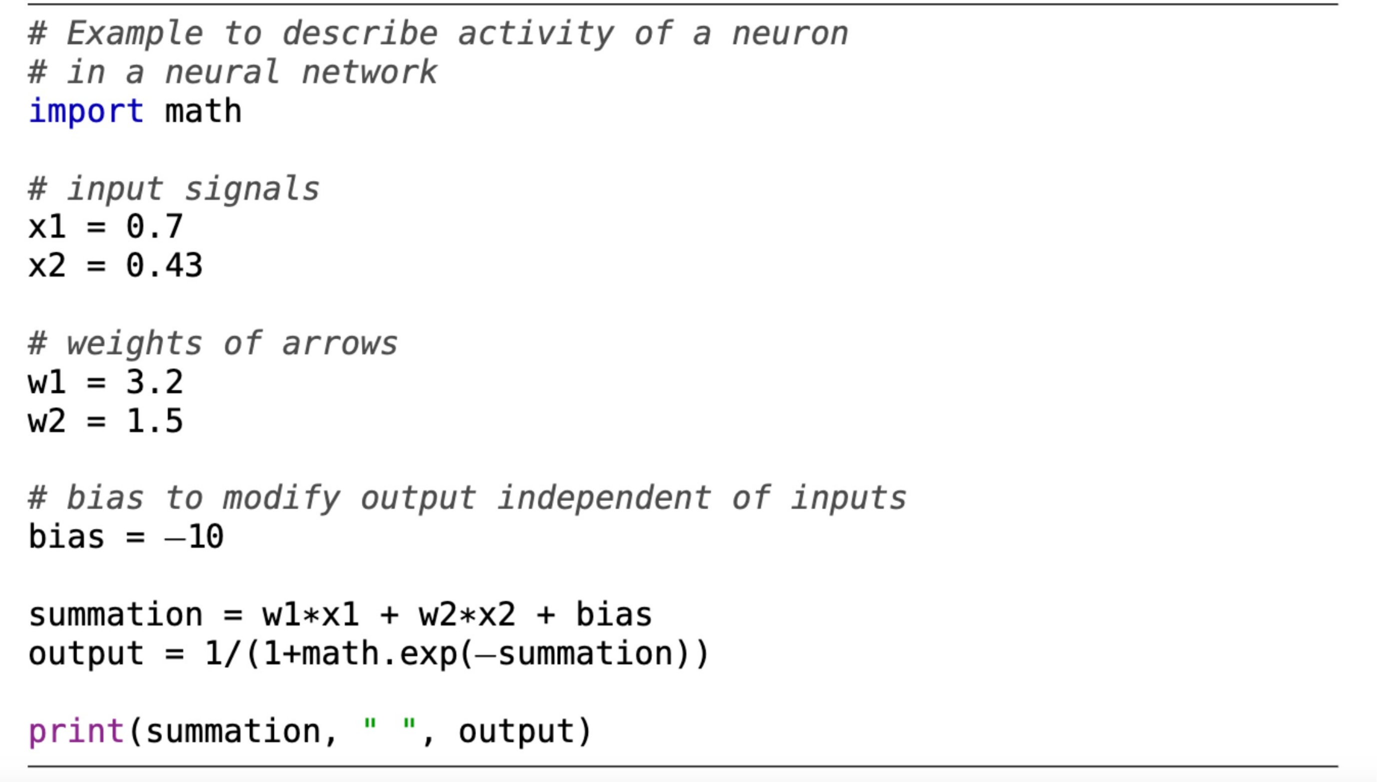
activation $f(x) = \frac{1}{1+e^{-x}}$
(sigmoid function)

(source: towardsdatascience.com)

- This is a single ‘neuron’
- It has four inputs, with four weights
- And a bias factor
- These are summed in the green node
- The sum is passed through an activation function
- Activation function is the sigmoid function
- Output is y_{pred}

Calculating a simple neuron, simply

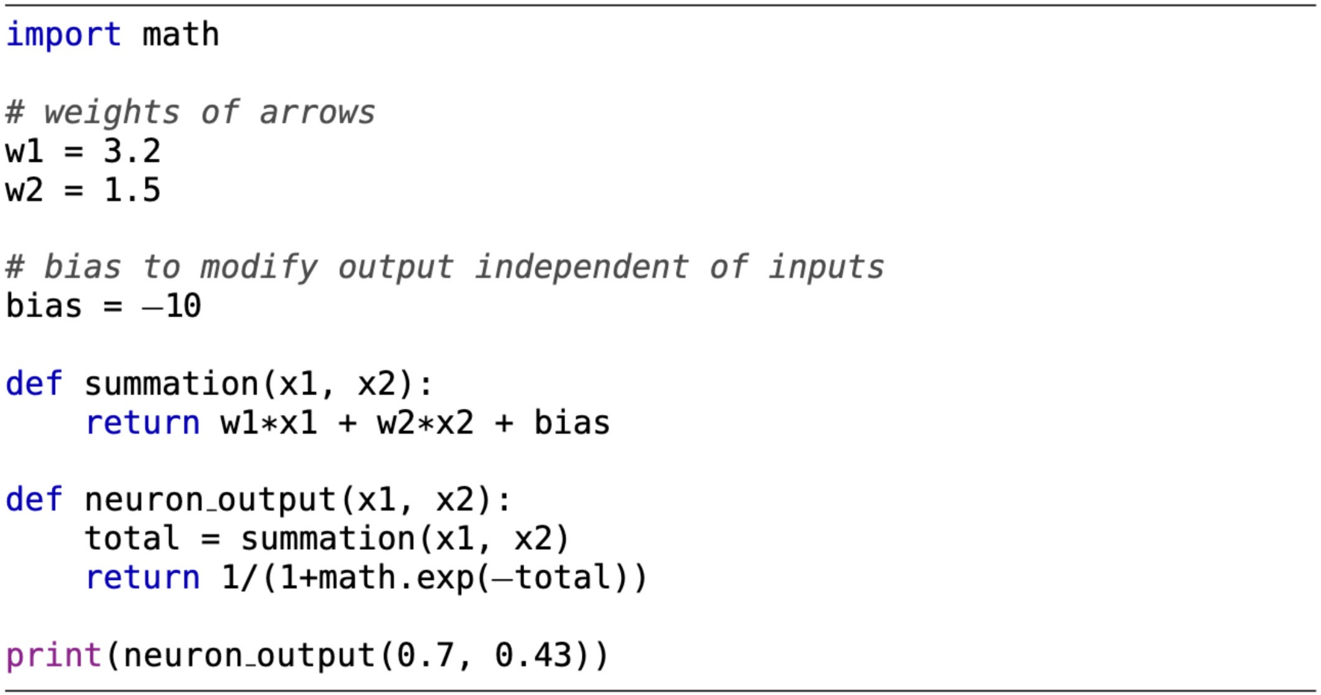




```
# Example to describe activity of a neuron
# in a neural network
import math

# input signals
x1 = 0.7
x2 = 0.43

# weights of arrows
w1 = 3.2
w2 = 1.5

# bias to modify output independent of inputs
bias = –10

summation = w1∗x1 + w2∗x2 + bias
output = 1/(1+math.exp(–summation))

print(summation, " ", output)
```

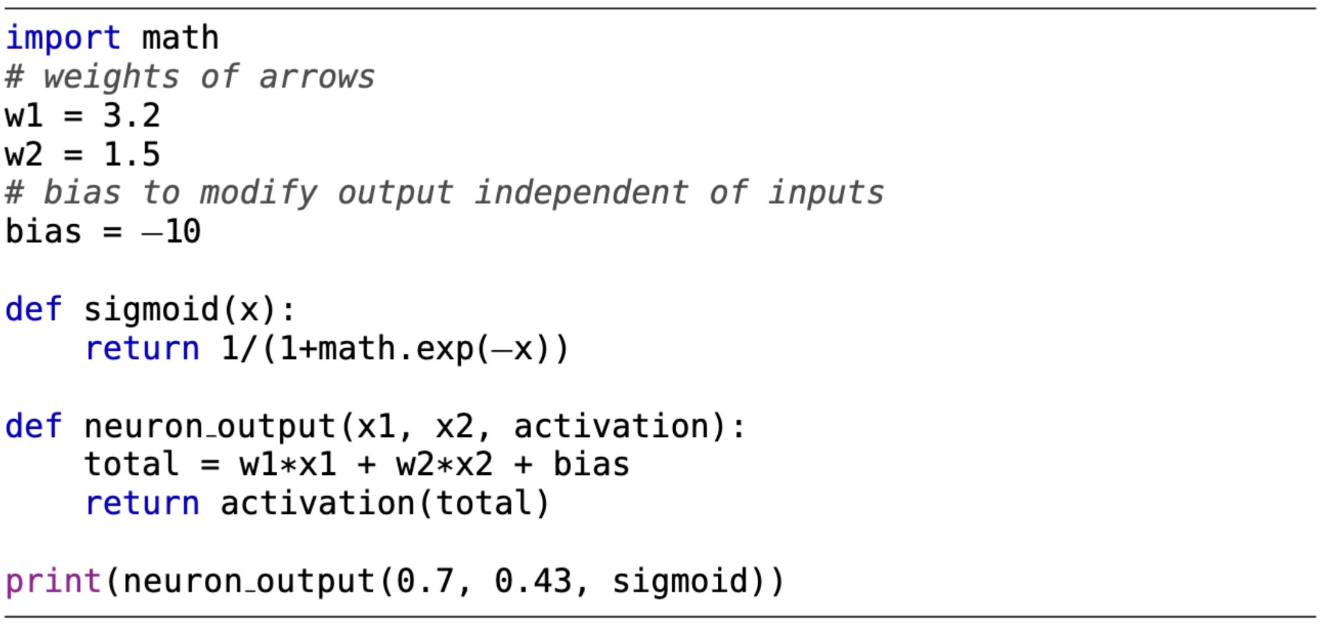
Re-writing to use functions

- Let’s try to recode this as a function that takes the inputs and produces the output of a single neuron



```
import math

# weights of arrows
w1 = 3.2
w2 = 1.5

# bias to modify output independent of inputs
bias = -10

def summation(x1, x2):
    return w1*x1 + w2*x2 + bias

def neuron_output(x1, x2):
    total = summation(x1, x2)
    return 1/(1+math.exp(-total))

print(neuron_output(0.7, 0.43))
```

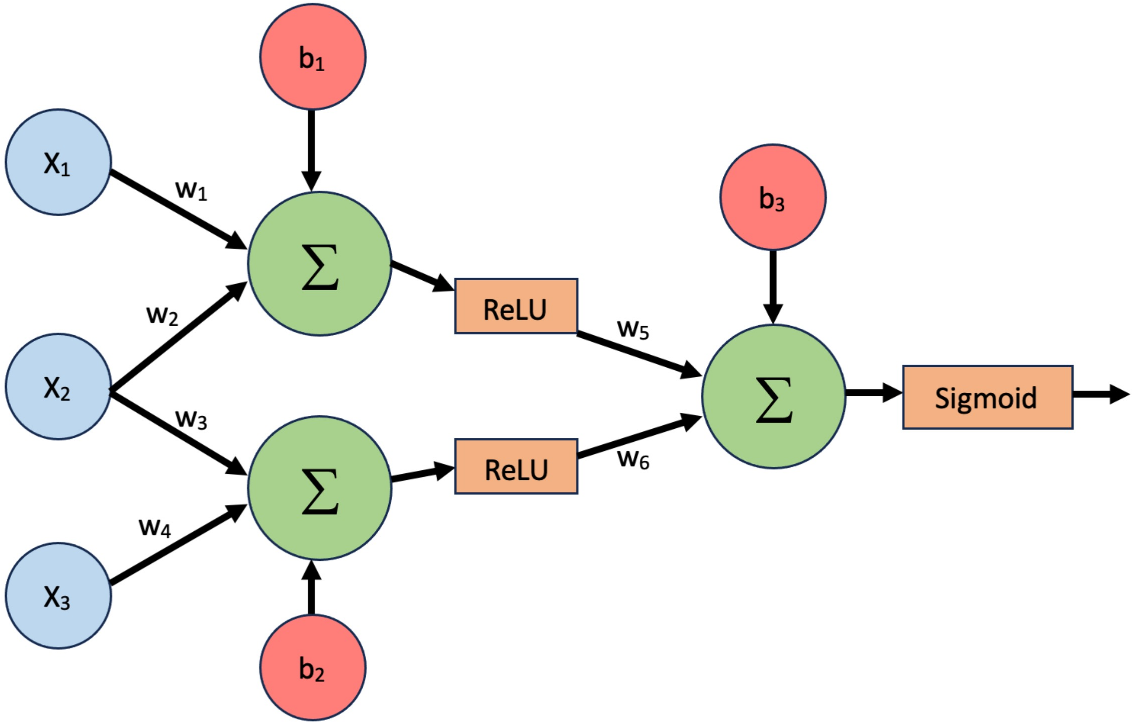
Passing functions to functions



```python
import math
# weights of arrows
w1 = 3.2
w2 = 1.5
# bias to modify output independent of inputs
bias = –10

def sigmoid(x):
    return 1/(1+math.exp(–x))

def neuron_output(x1, x2, activation):
    total = w1*x1 + w2*x2 + bias
    return activation(total)

print(neuron_output(0.7, 0.43, sigmoid))
```

Functional abstraction

- Increased abstraction makes this scalable to more complex networks:



Branching

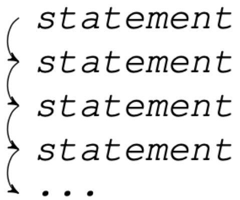
COMP1730/COMP6730

Reading: Textbook chapter 5 : Alex Downey, *Think Python*, 2nd Edition (2016) from *’Boolean expressions*’ to ‘*Nested conditionals*’

Australian
National
University

Program control flow

- **Sequential program execution:**

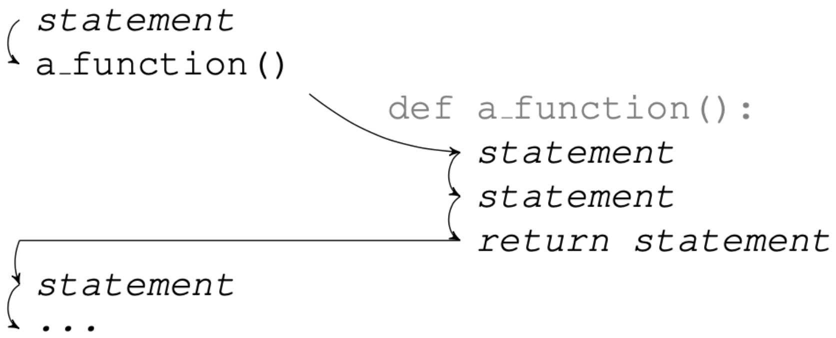


```
statement
statement
statement
statement
...
```

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

Program control flow

- **With functions and the stack:**

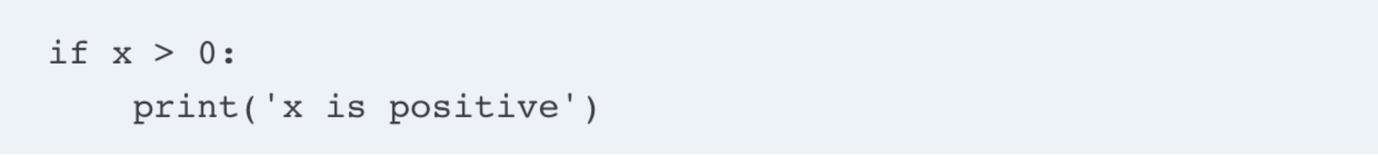


```
statement
a_function()
                    def a_function():
                        statement
                        statement
                        return statement
statement
...
```

- Function calls ‘insert’ a function body into this sequence, but the sequence of instructions remains invariably the same

Flow control: `if`

- The `if` statement evaluates whether a statement is `True` or `False`, then does something depending on the answer:



```
if x > 0:
    print('x is positive')
```

Expression is True

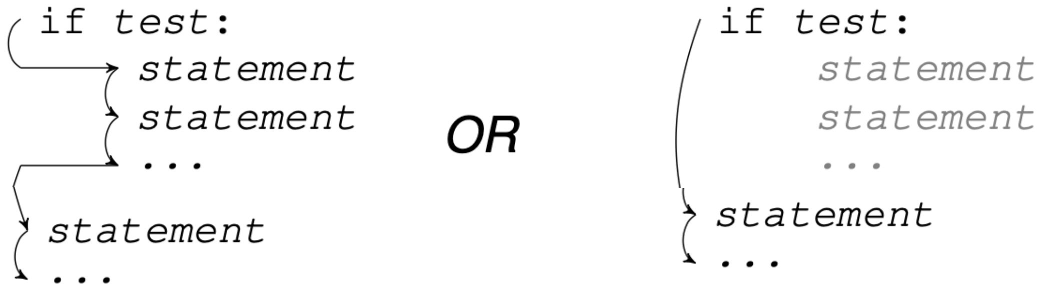
```
value = 1

if value > 0:
   # code block
   print(“Value is positive”)

# continue here
```

Expression is False

```
value = -1

if value > 0:
   # code block
   print(“Value is positive”)

# continue here
```

Branching program flow

- Depending on the outcome of a test, the program executes one of two alternative branches:



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

OR

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

Example

- The `if` statement

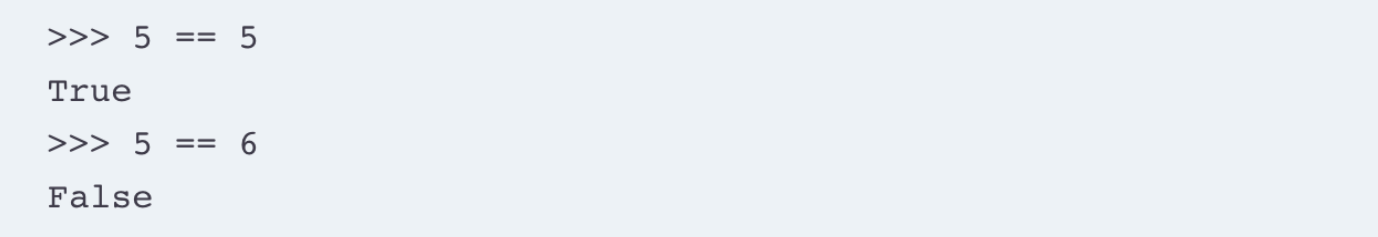
Code blocks (reminder)

- A block is a (sub-)sequence of statements
- A block must contain at least one statement
- In python, a block is delimited by indentation
    - All statements in the block **must be preceded by the same number of spaces/tabs** (standard is 4 spaces)
    - A block can include nested blocks (`if`'s, etc)
- Blocks with indentation are a python oddity
- (Almost) Every programming language has a way of grouping statements into blocks
    - For example, in C, Java and many others:

```
if (expression) {
  block
}
```

The ‘==‘ operator (reminder)

- Unlike the ‘=‘ operator, the ‘==‘ evaluates two values for equality
- The return value of this operator is a Boolean value, depending on the statement being evaluated



```
>>> 5 == 5
True
>>> 5 == 6
False
```

Downey (2015) Think Python, 2nd Ed.

Boolean expressions (reminder)

- A Boolean expression evaluates to either `True` or `False`. Note these are keywords in Python.
- A Boolean variable contains `True` or `False` values.
- Boolean **values** are returned by **comparison operators (==, !=, <, >, <=, >=)** and a few more
- **Boolean operators** (`and`, `or` and `not`) allow comparison of Boolean values (next slide)
- *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 (`+`, `=`, `*`, `/`) on Boolean values

Boolean operators

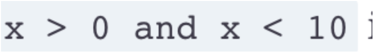
- The operators `and`, `or` and `not` combine truth values:

a `and` *b*	`True` if *a* and *b* both evaluate to `True`
a `or` *b*	`True` if at least one of *a* and *b* evaluates to `True`
`not` *a*	`True` if *a* evaluates to `False`

- Boolean operators have lower precedence than comparison operators (>, <, >=, <=, ==, !=) - which have lower precedence than arithmetic operators (*, /, +, -)

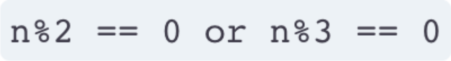
Chaining operators: `and`, `or` and `not`

- These logical operators are a means of chaining together logical statements:
- And:



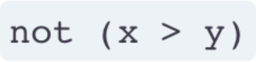
```
x > 0 and x < 10 i
```

- Or:



```
n%2 == 0 or n%3 == 0
```

- Not:



```
not (x > y)
```

- There are no limits to how these might be put together.

Example

- The `if` statement with chained operators

Back to `if`: alternative execution

- Sometimes called an ‘`if-else`’ statement:

Expression is True

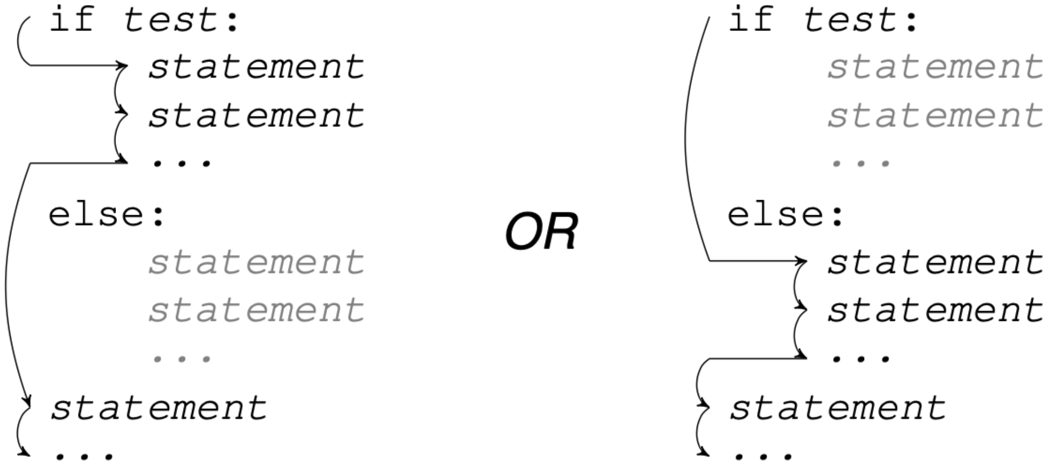
```
value = 34

if value % 2 == 0:
  # code block for True
  print(“Even number”)
else:
  # code block for False
  print(“Odd number”)

# continue here
```

Expression is False

```
value = 31

if value % 2 == 0:
  # code block for True
  print(“Even number”)
else:
  # code block for False
  print(“Odd number”)

# continue here
```

Branching program flow

- Depending on the outcome of a test, the program executes one of two alternative branches:



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

OR

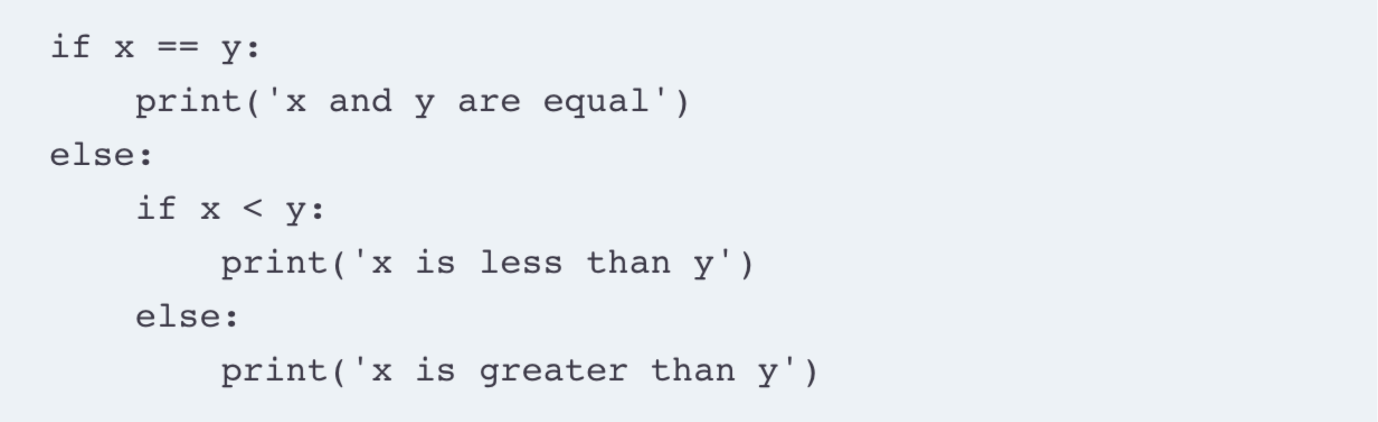
```
if test:
    statement
    statement
    ...
else:
    statement
    statement
    ...
statement
...
```

Example

- The `if-else` statement

Nested conditionals

- You can nest conditional statements within another conditional statements:

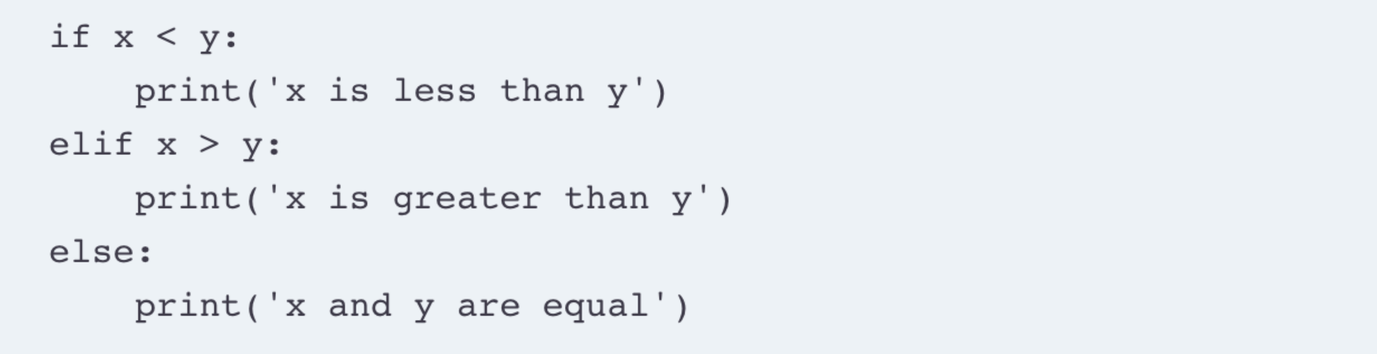


```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

Downey (2015) Think Python, 2nd Ed.

`elif`: switches

- And these can be chained together with `elif` to make ‘chained conditionals’:

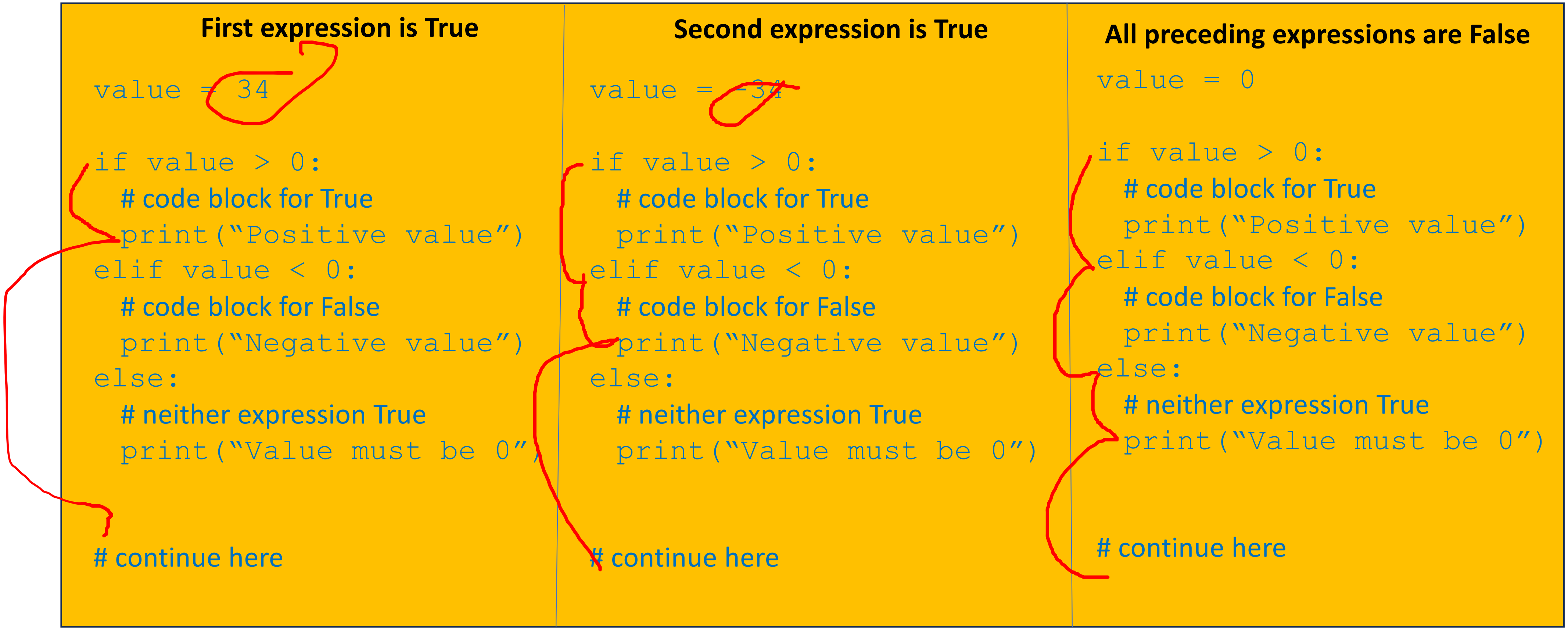


```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

Downey (2015) Think Python, 2nd Ed.

- When including an `else`, it must be at the end of the chain. But including a final `else` is optional

if-elif-else



First expression is True

```
value = 34

if value > 0:
   # code block for True
   print(“Positive value”)
elif value < 0:
   # code block for False
   print(“Negative value”)
else:
   # neither expression True
   print(“Value must be 0”)


# continue here
```

Second expression is True

```
value = -34

if value > 0:
   # code block for True
   print(“Positive value”)
elif value < 0:
   # code block for False
   print(“Negative value”)
else:
   # neither expression True
   print(“Value must be 0”)


# continue here
```

All preceding expressions are False

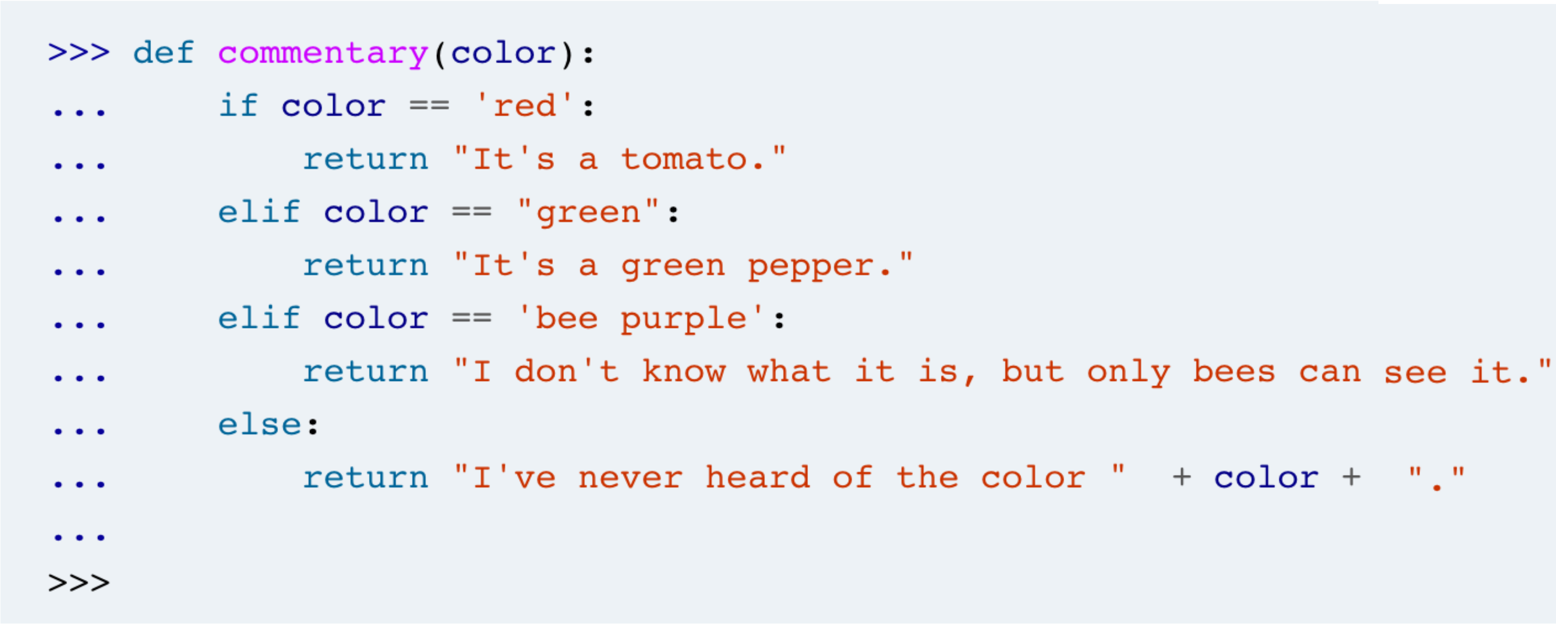
```
value = 0

if value > 0:
   # code block for True
   print(“Positive value”)
elif value < 0:
   # code block for False
   print(“Negative value”)
else:
   # neither expression True
   print(“Value must be 0”)


# continue here
```

Example

- The `if-elif-else` statement

Multiple `return` statements with `if`

- The `return` statement causes execution to leave the function block and return to where a function call was made
- There can be multiple `return` statements in a single function



```
>>> def commentary(color):
...     if color == 'red':
...         return "It's a tomato."
...     elif color == "green":
...         return "It's a green pepper."
...     elif color == 'bee purple':
...         return "I don't know what it is, but only bees can see it."
...     else:
...         return "I've never heard of the color "  + color +  "."
...
>>>
```

Lubanovic (2019) *Introducing Python*

Exercises

- Complete Exercises 5-1, 5-2 and 5-3 of *Think Python*.

Reading

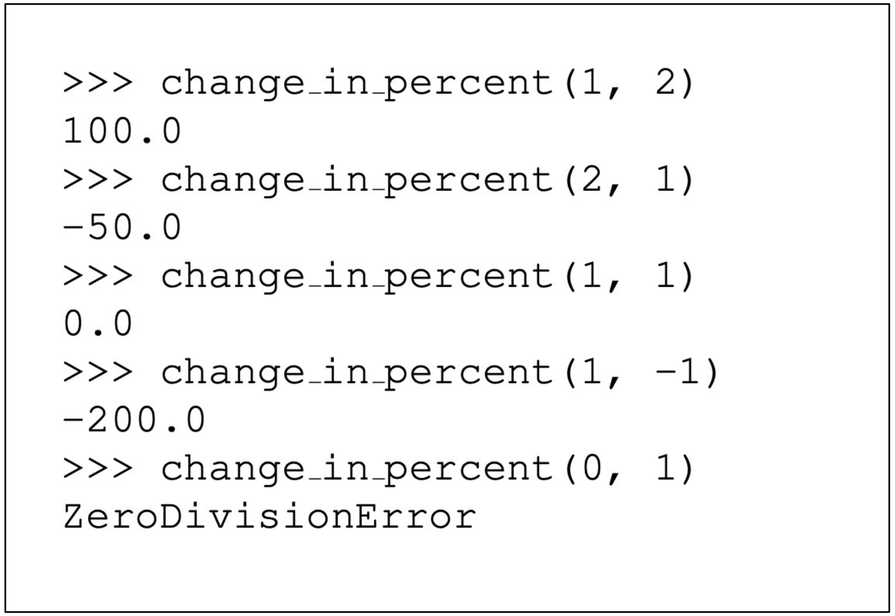
- Chapter 5 of *Think Python* from *’Boolean expressions*’ to ‘*Nested conditionals*’ AND/OR
- Section 4.2 of *Intro to Sci Prog with Python*

Testing and assertions

COMP1730/COMP6730

Function testing

- A function makes a logical unit for testing:
  - Documented input requirements
  - Expected output
- Testing can run a large variety of cases to ensure correct input produces expected output
- With lots of testing will identify edge-cases - try a range of typical input arguments:
  - values equal to/less than/greater than zero
  - very large and small values
  - values of equal and opposite signs



```
>>> change_in_percent(1, 2)
100.0
>>> change_in_percent(2, 1)
-50.0
>>> change_in_percent(1, 1)
0.0
>>> change_in_percent(1, -1)
-200.0
>>> change_in_percent(0, 1)
ZeroDivisionError
```

Testing code: `assert`

- Why is testing so important?
  - In a large code-base, tests keep a project within design parameters
  - Testing and fixing bugs can mean that routine code releases are _less_ stressful.
- Sanity checks find bugs introduced during development
  - Routine checking that developing one part of the codebase doesn't cause other parts to stop working
  - Or worse, silently start doing the wrong thing
- Testing that a function returns an expected value for standard input is common.
  - Basis of unit tests
- And, we use the `assert` statement to help mark your exams.

assert statement

Australian
National
University

- Syntax:

  assert expression, "assertion error message"
- An assertion performs a sanity check that something that should be True is actually True
- Unlike an if statement, assert will do nothing if the expression is True
- assert will only do something if the expression evaluated is False
  - What is does is raise and AssertionError !

if

```
value = 1

if value > 0:
print("Value is positive")
```

assert

```
value = ~~1~~ 2

assert value == 1, "Value not 1"
```

Example

- The `assert` statement

Assertions in the homework program

- `assert` is used to check if your homework calculates the correct values