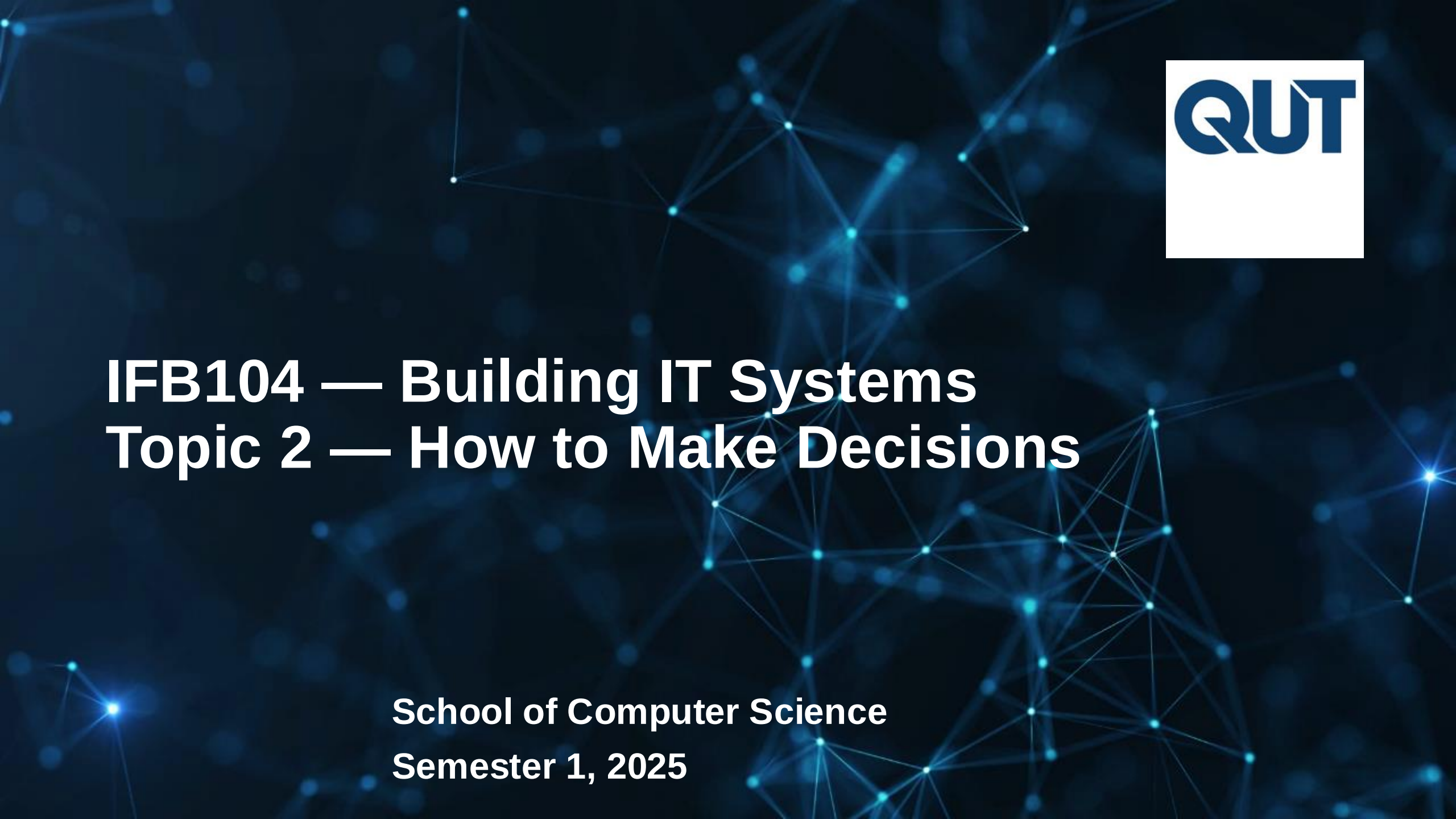


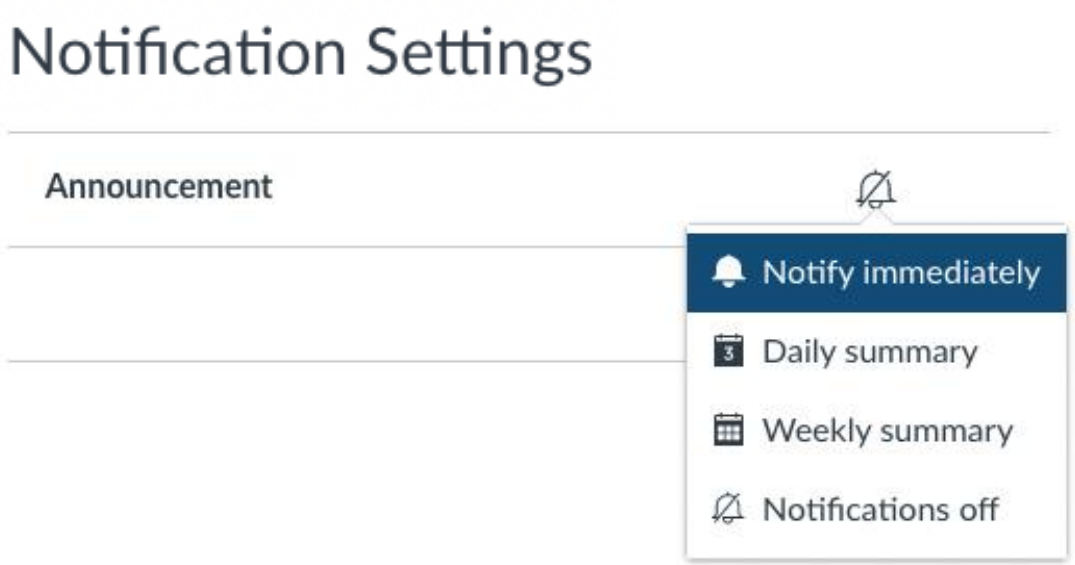
IFB104 — Building IT Systems
Topic 2 — How to Make Decisions

School of Computer Science

Semester 1, 2025

Housekeeping

- Instructions and code templates for Assessment Task 1A will appear on Canvas today in the *About Assessment* module
- Make sure you receive Canvas notifications for important announcements about assessment tasks



Assessment

- The requirement for Assessment Task 1a will appear on Canvas today. You are writing a program for a client who wants to be able to search through a list of books that is provided as a single piece of text.
  - But first then the client wants to see an initial installment by **Friday of Week 3** as a test of good faith.
  - For now, the client just wants to know if you can write an interactive program that calls some functions.
  - Just like in real life, the client doesn't always tell you everything you need to know straight away. Their contract requires you to deliver your final program by Friday of Week 5, and they may add requirements up until then.
- By the end of this week's workshop, you will know enough to complete Assessment Task 1A, so you can begin straight away.

Need help?

- The *Student Success Group* provides help to IFB104 students throughout the semester
  - Special workshops (additional to those offered in IFB104)
  - One-on-one consultation with a Peer Learning Facilitator (“STIMulate”)
  - Both online and on-campus
  - Some special workshops are already running and consultation services start this week
- See Canvas under *Unit Overview* | *Getting Help* for details

Get ready for maths, science and IT

Set yourself up for success this semester with STEM preparatory workshops for Faculty of Science students

Find free workshops that cover a range of topics including enhanced study skills, algebra, calculus, probability, Excel, Python, and more.

https://unihub.qut.edu.au/students/events/search?Text=ifb104

Get help and learn with peers

Drop by STEM peer support with STIMulate

The STIMulate Team are trained volunteer students. Many maths, science, IT and Engineering topics are supported through **free and confidential drop-in sessions**.

- Weeks 2 to 13, Monday to Friday
- Gardens Point Learning Hub, Level 2 of the Gardens Point Library (V block)

Check out the live roster right now

Come as often as you like. You'll find trained Peer Learning Facilitators who have excelled at your content and are volunteering their time to support your learning needs.

stimulate.qut.edu.au

STIMulate

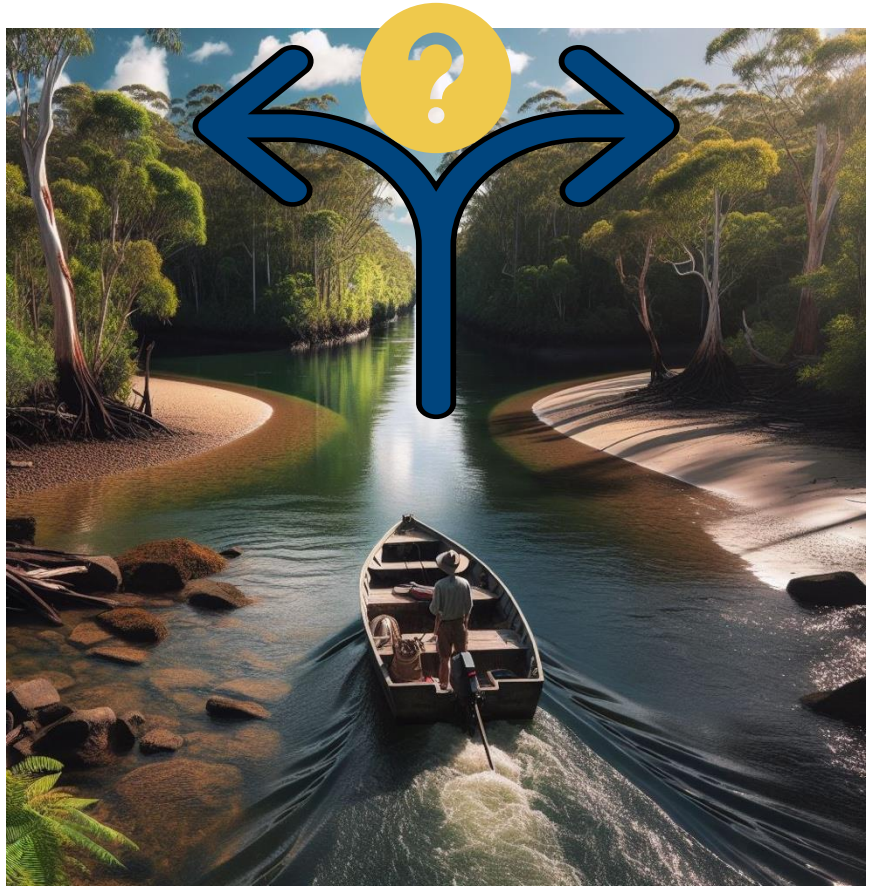
Facilitating real success in maths, science and IT across QUT

The challenge of building IT systems

- Information Technology is about getting a computer to do some work for us
- But computers are stupid!
  - They can only do what they're told
  - And they can do simple actions only
- Their power is due to their ability to repeat pre-defined functions very *quickly*

This week

- How to use and create your own functions in Python
- Boolean-valued expressions as the basis for making decisions in computer programs
  - Boolean operators for constructing truth-valued expressions
- Conditional statements
  - The `if` statement
  - The `if`-`else` statement
  - The `if`-`elif`-`else` statement
  - Nested `if` statements



Part A — Expressions, variables and functions

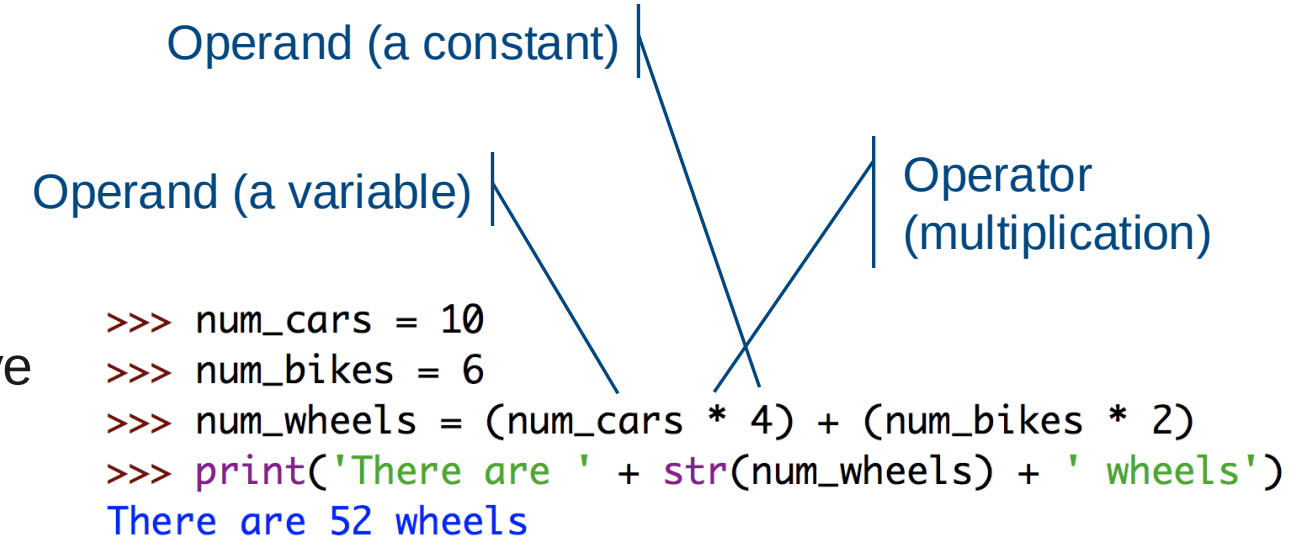
Key programming concept: Three d ways of writing expressions

- Confusingly, a programming language such as Python offers three distinct ways of performing operations on data values:
  - operators
  - **functions**
  - methods
- There are sound historical and technical reasons for this, but they are not always apparent to the beginner programmer!

```
>>> days = 7
>>> days # There are seven days in a week
7
>>> days - 2 # But let's exclude weekends
5
>>> days.__sub__(2) # What the heck is this?
5
>>> # It's another way to do subtraction,
>>> # but it sure looks weird!
```

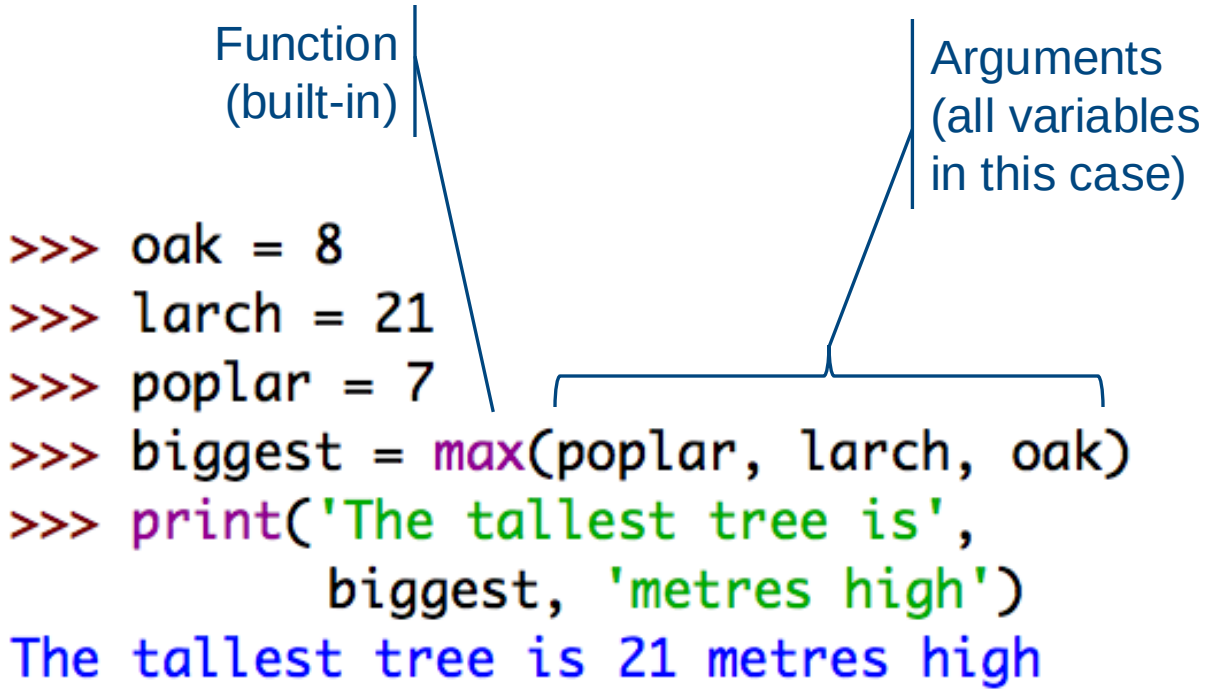
Built-in operators

- We have seen that we can perform various operations on data values, to produce new values, using familiar arithmetic symbols
  - The data values may be expressed directly, as *literal constants*, or may have been stored previously in a named *variable*
  - However, only a small number of operations have special symbols like '*' and '+' because we don't have very many special characters on our keyboard!



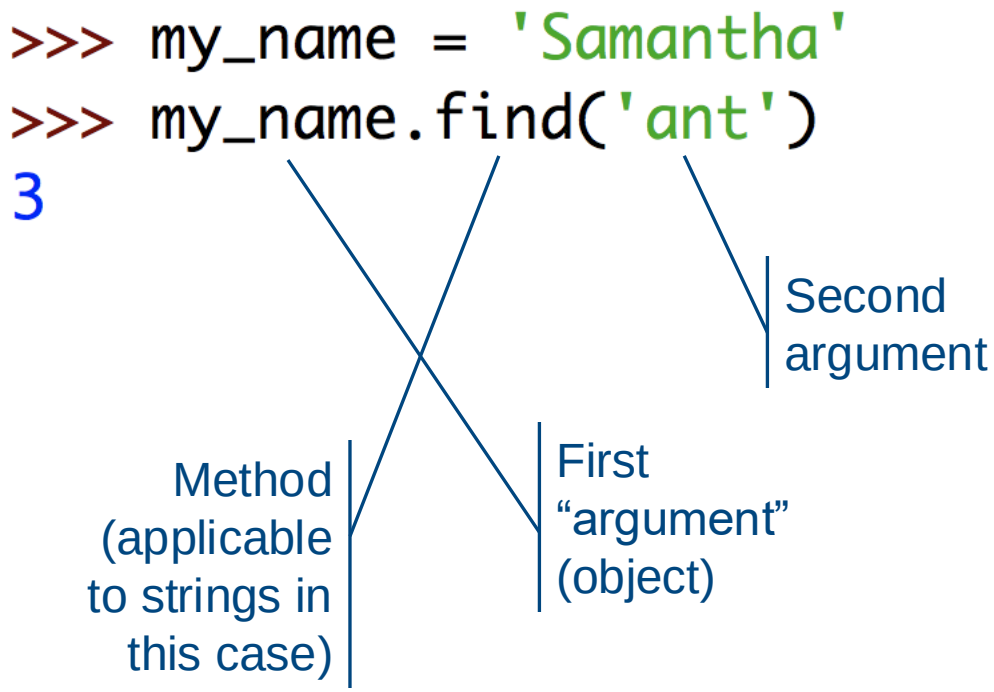
Functions

- Named functions are called by following their name with comma-separated arguments enclosed in round brackets
  - The brackets must appear in order to call the function even if no arguments are supplied



Methods

- “Methods” are special functions applicable to values of certain types (“classes”) and are called with the first argument *preceding* the function name
  - This is sometimes called “dot notation”
- Different types (classes) of values (objects) have different methods applicable to them



Key programming concept: Two reasons for calling a function (or method)

- Most functions/methods accept some arguments and *return* a new value, which can be used in a larger expression, assigned to a variable, or printed to the screen
- However, some functions produce a *side-effect* on variables or the computing environment and return nothing
  - An attempt to access the value returned by a pure side-effecting function gets the special value `None`
- And some functions do *both* (e.g., Turtle's `stamp` method both draws an image and returns an identifier)

```
>>> # import a constant from a module
>>> from math import pi
>>> # evaluate an expression and store the
>>> # value returned
>>> area = pi * (4 ** 2)
>>> area
50.26548245743669
>>>
>>> # define a list of values
>>> directions = ['up', 'down', 'left']
>>> # apply a function to the list that returns
>>> # a result
>>> len(directions)
3
>>> # apply a method to the list that has a
>>> # side-effect
>>> directions.remove('left')
>>> directions
['up', 'down']
```

Optional and named arguments

- Some functions, like the built-in `round` function, allow certain arguments to be optional
- Some functions, like the built-in `sorted` function, allow us to specify which arguments we wish to provide by their name

```
>>> weight = 67.8934 # define a floating point number
>>> round(weight) # return it rounded to a whole number
68
>>> round(weight, 2) # specify how many digits to return
67.89
>>>
>>> colours = ['red', 'green', 'blue'] # define a list
>>> sorted(colours) # return it sorted alphabetically
['blue', 'green', 'red']
>>> sorted(colours, reverse = True) # sort backwards
['red', 'green', 'blue']
>>>
```

More character string operations

- We have seen that simple numeric *operations* such as `+`, `*`, etc, work with character strings
- Some other commonly-used string operations are shown on the right
- All of these operations *return* a new string, leaving the original one unchanged
  - If you want to keep the new string you need to assign it to a variable
- There are many, many more!
  - See the *Python Standard Library* manual under *Built-in Types* | *Text Sequence Type* for more details

```
>>> phrase = 'Hello there, World!!!'
>>> len(phrase) # length of a string
21
>>> phrase.replace('Hello', 'Goodbye') # replace substring
'Goodbye there, World!!!'
>>> phrase[1] # get the char at position 1
'e'
>>> phrase[6:11] # substring from positions 6 upto 11 (exclusive)
'there'
>>> phrase.split() # split the string on spaces
['Hello', 'there,', 'World!!!']
>>> phrase.strip('!') # remove leading/trailing characters
'Hello there, World'
```

Two ways of inserting variable values into strings

- We've already seen that we can concatenate string values using the '+' operator
- Another way to insert values into strings is to use *formatted literal strings*, or 'f-strings' for short

```
>>> my_name = 'George'
>>>
>>> # We can add strings together
>>> print('Just call me ' + my_name + '.')
Just call me George.
>>>
>>> # Or we can insert variable values
>>> # into 'formatted' strings
>>> print(f'Just call me {my_name}.')
Just call me George.
```

Nice formatting of numbers

- When preparing to print long floating-point numbers we can modify them by deleting the fractional part (using `int`) or rounding them (using `round`)
  - Leading and trailing zeros are not shown
- To achieve more precise control over how many digits are shown we can use the string `format` method
  - The string to be printed can contain numbered placeholders for values, starting at 0
  - The placeholder can specify formatting, especially the number of digits after the decimal point

```
>>> # Calculate a floating point value
>>> item = 'Mark V Widget'
>>> unit_cost = 21.3456
>>> number_required = 5
>>> price = number_required * unit_cost
>>> price
106.72800000000001
>>>
>>> # Print it nicely in three different ways
>>> print(item, 'invoice:', round(price, 2), 'dollars')
Mark V Widget invoice: 106.73 dollars
>>> print(item + ' invoice: ' + str(round(price, 2)) + ' dollars')
Mark V Widget invoice: 106.73 dollars
>>> print('{0} invoice: {1:.2f} dollars'.format(item, price))
Mark V Widget invoice: 106.73 dollars
>>>
```

Text-based input and output

- One reason for needing to make choices is to decide how to respond to some form of *external input*
- The simplest form of input and output is text-based, using a computer keyboard and screen, respectively
- We have already seen that Python's `print` function displays text on the screen:

```
>>> name = 'Daria'
>>> print('Hello, ' + name)
Hello, Daria
>>>
```

Keyboard input

- Python provides two functions for getting keyboard input
- Function **`input`** returns whatever is entered on the keyboard as a string
  - It accepts an optional parameter which is displayed on the screen as a prompt
- Function **`eval`** returns the value of a Python expression appearing in a string
  - But an error occurs if the string cannot be interpreted as a valid Python expression

```
# Get some input from the user
response = input('Please enter an expression: ')
# Echo it
print("You typed '" + response + "'")
# Display its value
print('Your expression equals', eval(response))
# Display its type
print('Your expression is of type',
      type(eval(response)))
```

```
Please enter an expression: 6 * 7.2
You typed '6 * 7.2'
Your expression equals 43.2
Your expression is of type <class 'float'>
```

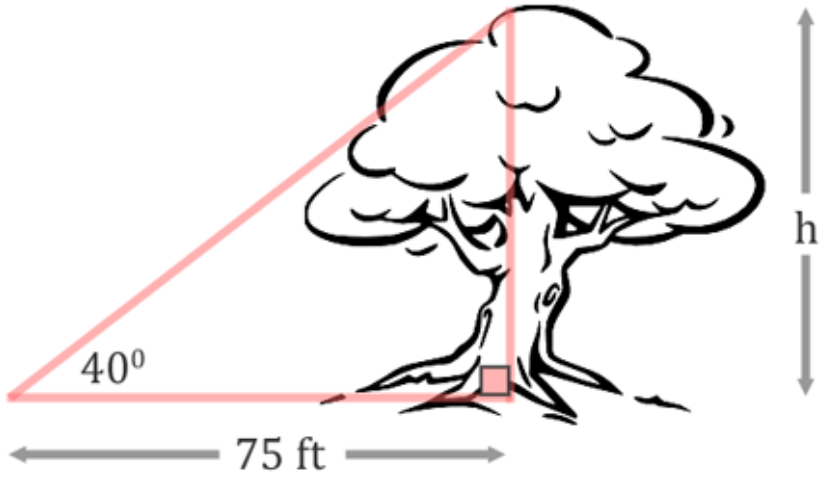
Functions and methods: instruct once, do many times (with a twist)!

- ”Computer” used to be a profession!
- Instructions are learned ahead, repeated with new entries for appropriate outputs
- The same set of instructions can be used by several computers, several times, but does not have to be spelled out every time.

https://www.flickr.com/photos/internetarchivebookimages/14570000517/

Code reuse

- In a number of examples we saw that the same or similar code was repeated several times
- Rather than duplicating similar code segments, it would be better to write the code once and merely invoke it whenever needed
  - Makes our programs smaller and more understandable
  - Makes our programs easier to modify and maintain
- Functions and modules are the standard mechanisms for creating *reusable* code




```
>>> distance_to_tree = 75 # feet
>>> angle_to_top = 40 # degrees
>>> from math import tan, radians # trig functions
>>> height_of_tree = \
    distance_to_tree * tan(radians(angle_to_top))
>>> print('The tree is', round(height_of_tree),
          'feet high')
The tree is 63 feet high
>>>
```

Key programming concept: Functions

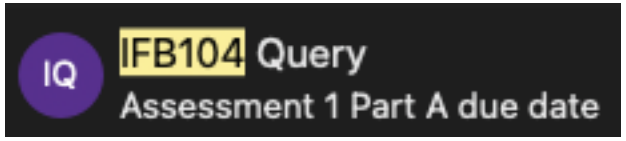
- A *function* is a *named*, *parameterised* sequence of statements
  - We can refer to a function by its name to invoke or “call” it
  - The statements forming the function’s “body” determine what it does
  - The parameters allow us to customise the function’s behaviour when we call it
  - Functions must be *defined* before they can be *called*
- Groups of related functions can be packaged into modules or “Application Programming Interfaces”

```
>>> # A function for computing heights
>>> def height(distance, angle):
        from math import tan, radians
        height_of_object = \
          round(distance * tan(radians(angle)))
        return height_of_object

>>> # Find the height of a tree
>>> distance_to_tree = 75 # feet
>>> angle_to_top = 40 # degrees
>>> print('The tree is',
          height(distance_to_tree, angle_to_top),
          'feet high')
The tree is 63 feet high
```

Example of defining a function

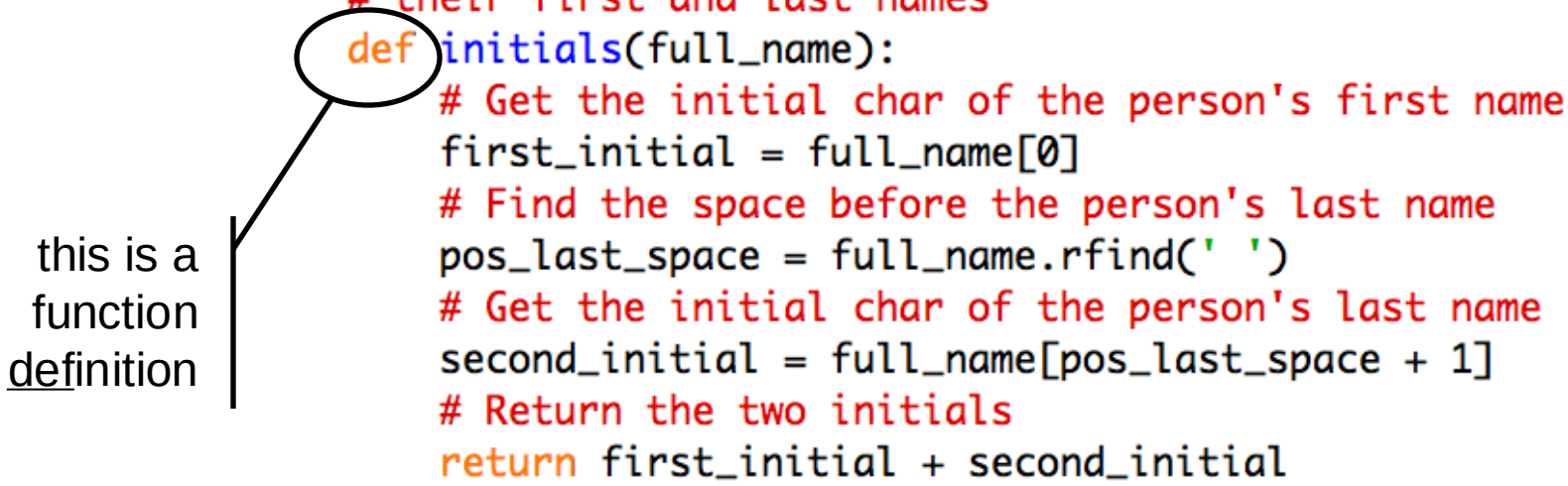
- Microsoft's *Outlook Web App* displays a little image of the sender next to the message, if the sender has created one
- However, if the sender hasn't created an image to display then *OWA* displays the person's initials based on their first name and surname, e.g.:
  - If the sender was `'James Bond'` it displays `'JB'`
  - If the sender was `'James Tiberius Kirk'` it displays `'JK'`



- Since this is a frequently-occurring task it's a good candidate for defining as a reusable function, rather than duplicating the code to extract the initials from someone's name everywhere it's needed
- Even if this code appears only once in the app it would still be a good idea to make it a function to better structure the program, making it easier to understand

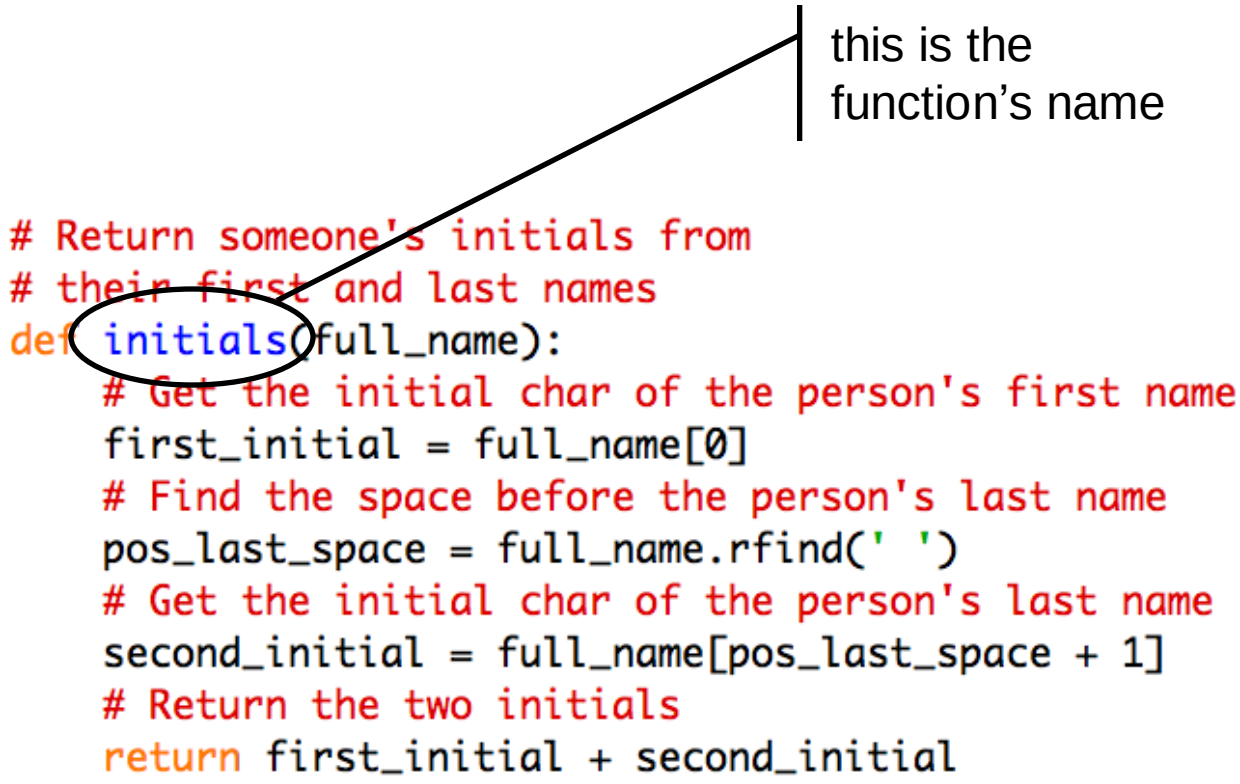
Example of defining a function

- A *function definition* introduces a new function with a *name*, optional *parameters* and a *body*



Example of defining a function

- A *function definition* introduces a new function with a *name*, optional *parameters* and a *body*



Example of defining a function

- A *function definition* introduces a new function with a *name*, optional *parameters* and a *body*

these are the function's parameters (just one in this case)

```
# Return someone's initials from
# their first and last names
def initials(full_name):
    # Get the initial char of the person's first name
    first_initial = full_name[0]
    # Find the space before the person's last name
    pos_last_space = full_name.rfind(' ')
    # Get the initial char of the person's last name
    second_initial = full_name[pos_last_space + 1]
    # Return the two initials
    return first_initial + second_initial
```

Example of defining a function

- A *function definition* introduces a new function with a *name*, optional *parameters* and a *body*

```
# Return someone's initials from
# their first and last names
def initials(full_name):
    # Get the initial char of the person's first name
    first_initial = full_name[0]
    # Find the space before the person's last name
    pos_last_space = full_name.rfind(' ')
    # Get the initial char of the person's last name
    second_initial = full_name[pos_last_space + 1]
    # Return the two initials
    return first_initial + second_initial
```

this is the function's body (an indented sequence of statements)

Example of defining a function

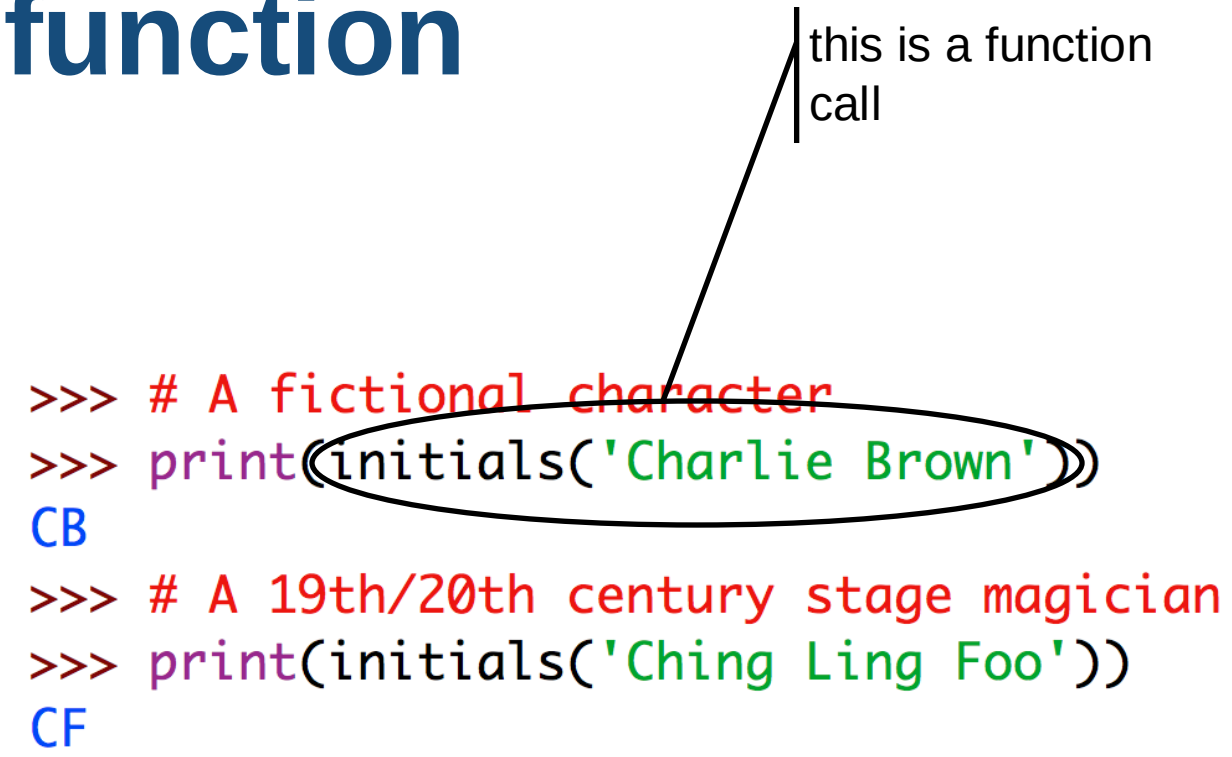
- A *function definition* introduces a new function with a *name*, optional *parameters* and a *body*

```
# Return someone's initials from
# their first and last names
def initials(full_name):
    # Get the initial char of the person's first name
    first_initial = full_name[0]
    # Find the space before the person's last name
    pos_last_space = full_name.rfind(' ')
    # Get the initial char of the person's last name
    second_initial = full_name[pos_last_space + 1]
    # Return the two initials
    return first_initial + second_initial
```

this is the value returned by the function (side-effecting functions don't have this)

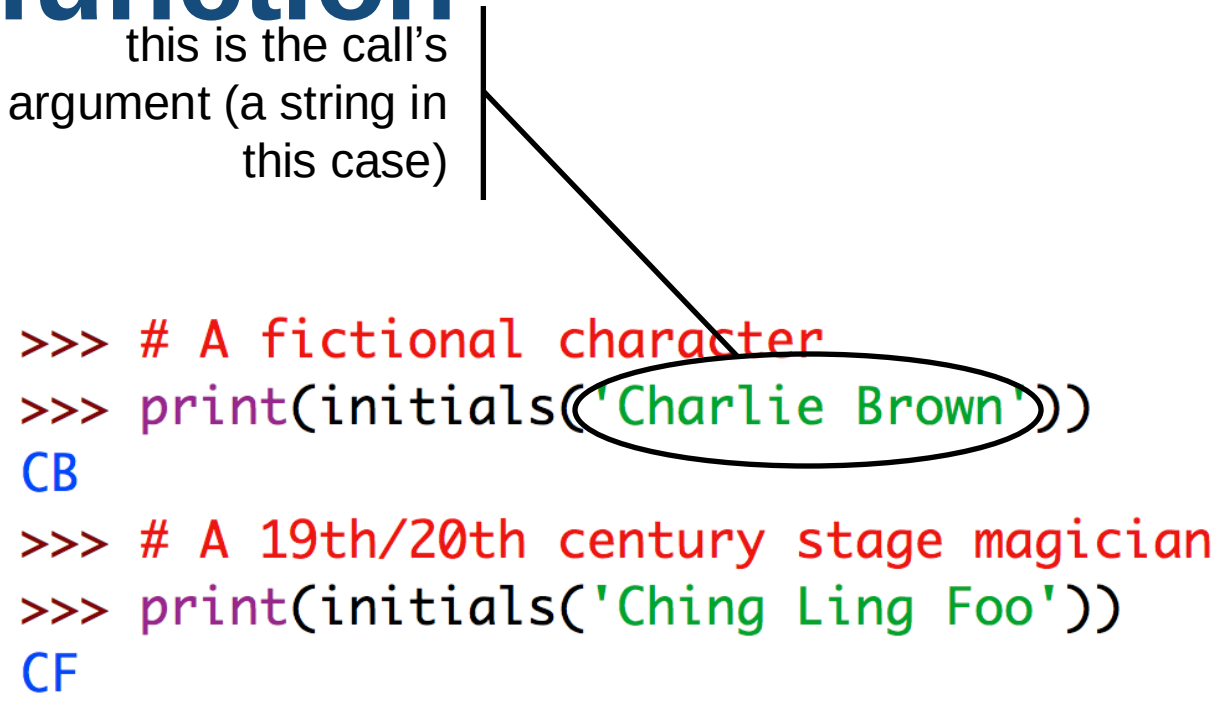
Example of calling a function

- To invoke a function that has been defined, simply use its name as a statement, if it has a side effect, or use it in an expression, if it returns a value
- An *argument* expression must be supplied for each of the function's parameters



Example of calling a function

- To invoke a function that has been defined, simply use its name as a statement, if it has a side effect, or use it in an expression, if it returns a value
- An *argument* expression must be supplied for each of the function's parameters



Key programming concept: Returning values

- Some functions are called for their side effects only, e.g., to draw some graphics on the screen
- Other functions are defined to perform a calculation on some data and return a result
- A **`return`** statement in a function's body determines what value the function returns when it is called
- A common mistake when coding in Python is to confuse merely printing a value with returning one
  - Both actions *look* the same in IDLE's shell interpreter because it *prints* returned values!

```
# Add emphasis and print the result
def exclaim_1(phrase):
    print(phrase + '!')

# Add emphasis and return the result
def exclaim_2(phrase):
    return phrase + '!'
```

```
>>> # Can use a returned value in an expression
>>> exclaim_2('Hello') + '?'
'Hello!?'
>>> # But this function call returns nothing
>>> exclaim_1('Hello') + '?'
Hello!
Traceback (most recent call last):
  File "<pyshell#125>", line 1, in <module>
    exclaim_1('Hello') + '?'
TypeError: unsupported operand type(s) for +: '
```

Key programming concept: Parameter passing

- To make functions customisable to different situations we give them *parameters*
- A function can have zero, one or many parameters
  - The parameter names used in the function definition are the *formal parameters*
  - The values or expressions used in the function call are the *actual parameters*, also called *arguments*
- Calling a function is like running the function's body with each occurrence of the parameters in the code replaced with the corresponding arguments

formal parameter

```
>>> tongue_twister = "Upon a slitted sheet I sit"
>>>
>>> # Return a chosen word in a phrase
>>> def word(index, phrase):
        words = phrase.split()
        return(words[index])

>>> # Find the fourth word (counting from zero)
>>> tongue_twister.split()[3]
'sheet'
>>>
>>> # Find the fourth word (counting from zero)
>>> word(3, tongue_twister)
'sheet'
```

argument

Default parameter values

- As well as specifying parameters that must be provided to call a function, we can also make some parameters optional by giving them default values
- Many of Python's pre-defined functions do this

```
# Raise the base to the given exponent,
# or square it by default
def power(base, exponent = 2):
    return base ** exponent
```

```
>>> power(4)
16
>>> power(4, 3)
64
>>>
```

Named parameters

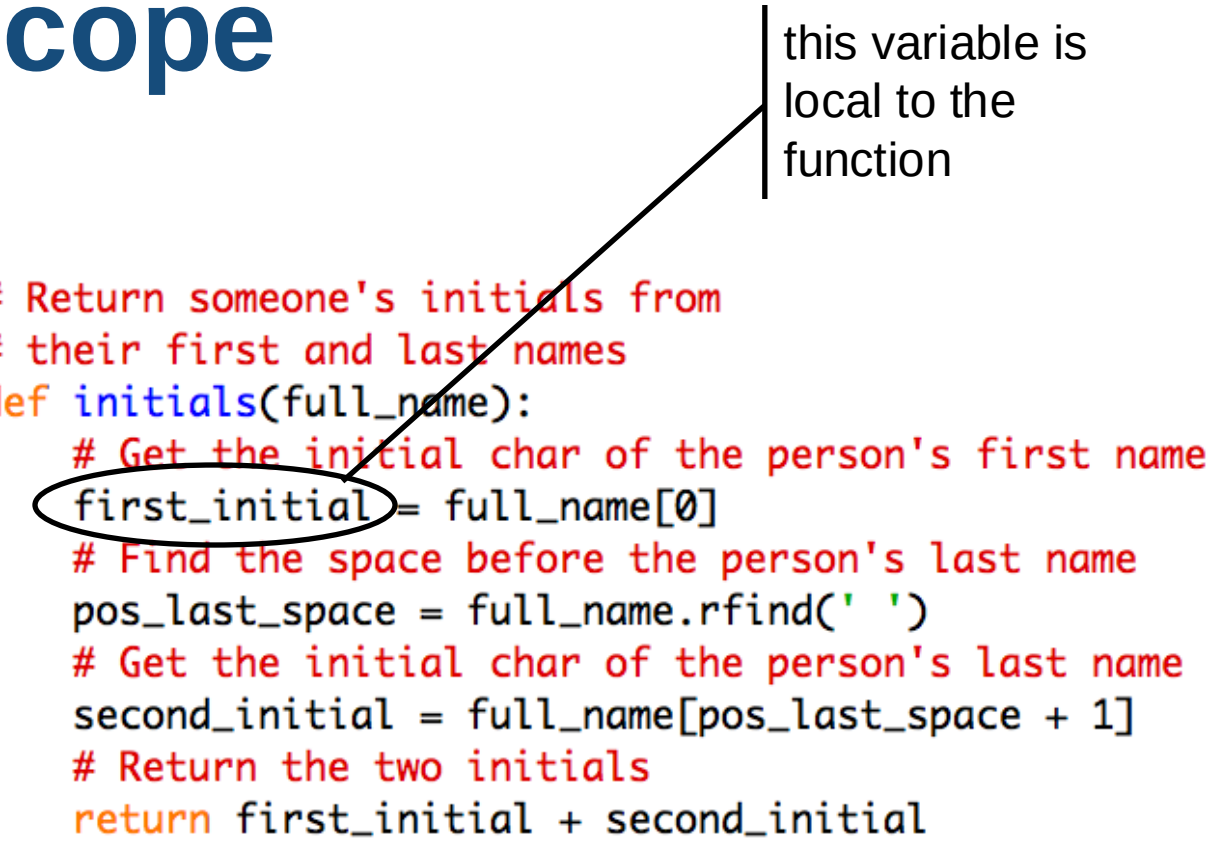
- Python supports both positional and named ("keyword") parameters
- Named parameters are convenient when a function has many parameters and you don't want to provide values for them all
- Many of Python's pre-defined functions work like this

```
# Raise the base to the exponent,
# with base 10 as default
def power(base = 10, exponent = 2):
    return base ** exponent
```

```
>>> # use default base
>>> power(exponent = 4)
10000
>>>
```

Local variables and scope

- Sometimes a function may require a variable to store a temporary value as part of its job
- Introducing a new variable within a function's body creates a *local variable*
- Local variables and parameters are not accessible to statements outside the function body
  - The *scope* of the "locals" is limited to within the function
- To *assign* to a global variable from within a function we need to declare it as `global` (otherwise it would be treated as a new local variable)



Demonstration files: `scoping.py`, `global_variable.py`, `ref_transparency.py`

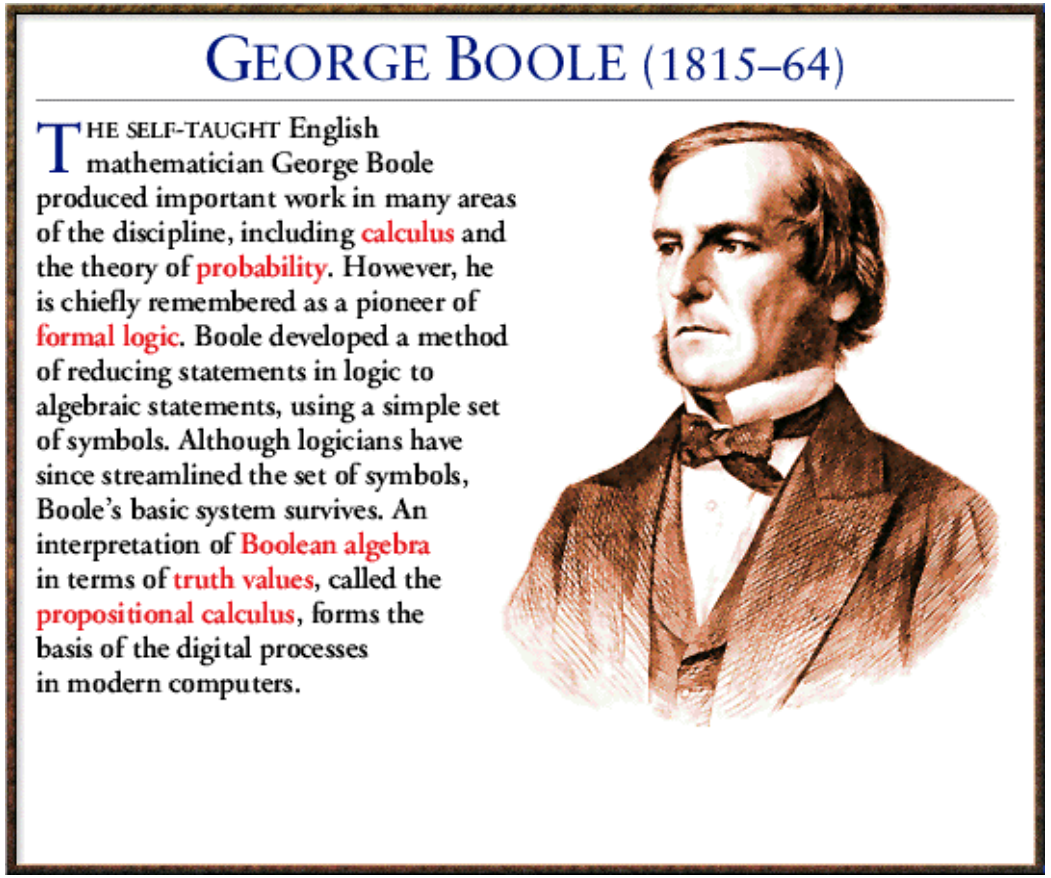
Choose your names with care

- Although you usually have a free choice of variable, function and module names, all programming languages have “reserved words” or “keywords” that shouldn’t be used for any other purpose
- Similarly, it’s unwise to use the names of built-in functions or standard modules for other purposes
  - Your program may fail in mysterious ways if you do!
- In Python this means we shouldn’t use certain names for our own variables, functions and modules, including:
  - Statement keywords such as `for`, `def`, `return`, etc
  - Built-in function names such as `range`, `max`, `len`, etc
  - Standard module names such as `math`, `turtle`, etc

Part B — Boolean Expressions

The Boolean type

- The Boolean type (named after 19th-century British mathematician George Boole, hence the capital ‘B’) is based around the two literal values `True` and `False`
- Boolean expressions, i.e., those that return a truth value, are used so we can choose between alternative actions in programs



Key programming concept: Relational operators

- Boolean expressions involving numeric values typically use *relational operators*
- Each of these expressions will return `True` or `False`
- Some of these operators work on strings too

Expression	Meaning
x > *y*	Is *x* greater than *y*?
x < *y*	Is *x* less than *y*?
x >= *y*	Is *x* greater than or equal to *y*?
x <= *y*	Is *x* less than or equal to *y*?
x == *y*	Is *x* equal to *y*?
x != *y*	Is *x* not equal to *y*?
x **in** *y*	Does *x* occur in *y*? (Where y is a compound type such as a string or list)

Examples of relational expressions

- After this sequence of assignments …

```
a = 4
b = 6
c = 5
```

- These Boolean expressions evaluate to **`True`**

```
b >= a
a != c
a < c
b <= 10
```

- And these evaluate to **`False`**

```
a >= 5
b <= a
b == c
```

A note on syntax

- Note the difference between '**=**' and '**==**'
- Assignment to a variable uses **=**
- Tests for equality use **==**
- This confusing choice of operators is used by most modern computer languages (unfortunately!)

```
>>> weight = 64 # an assignment
>>> weight == 64 # an equality test
True
>>> 57 == weight # another test
False
>>> 57 = weight # an illegal assignment
SyntaxError: can't assign to literal
```

Key programming concept: Boolean connectives

- In everyday speech we often combine a number of true/false statements
  - "It's hot **and** it's sunny"
  - "I'm **not** tired"
  - "We can go to the beach **or** we can do our homework"
- In computer languages these same *connectives* can be used to combine Boolean-valued expressions

- Assume that *B* and *C* are Boolean-valued expressions
- *B* `and` *C*
  - Returns `True` if both *B* and *C* are `True`
  - Returns `False` if either *B* or *C* is `False`
- *B* `or` *C*
  - Returns `True` if either *B* or *C* is `True`
  - Returns `False` if both *B* and *C* are `False`
- `not` *B*
  - Returns `True` if *B* is `False`
  - Returns `False` if *B* is `True`

Examples of Boolean expressions

- After this sequence of assignments …

```
a = 4
b = 6
c = True
```

- These Boolean expressions evaluate to **`True`**

```
a == 4 and b < 10
not (a == b)
b < 10 and a < 10
c or a == 9 # This one is tricky!
```

- And these evaluate to **`False`**

```
a == 4 and b == 7
b > 10 or a > 10
not c
```

Boolean operator precedence and ordering

- Recall that in mathematics arithmetic operators have a *precedence* order:

$$4 * 5 + 3 = 23 \quad \text{but} \quad 4 * (5 + 3) = 32$$

- Boolean operators also follow a particular precedence
  - Whenever there is any doubt, use brackets to avoid ambiguity!
  - Also note that in most programming languages Boolean expressions are evaluated from left to right

- Imagine that we want to give concession tickets to all students and seniors, but only on weekends:

```
>>> weekend = False # it's not a weekend
>>> student = False # person is not a student
>>> senior = True # person is a senior
>>> # Should we give the person a concession?
>>> weekend and student or senior # Wrong answer!
True
>>> weekend and (student or senior) # Right answer
False
```

Key programming concept: Predicates (Boolean-valued functions)

- Functions that return Boolean results can be used to control the choice of actions in Python programs
  - Functions that return Boolean values are called *predicates*
  - You can create your own predicates simply by defining functions which return `True` or `False`
- A well-structured, easy-to-read and maintain program will typically define a function for complex Boolean expressions, rather than writing the full expression every time it is used

```
# Function to tell us if the first
# sequence is longer than the second
def longer(seq_a, seq_b):
    return len(seq_a) > len(seq_b)
```

```
>>> word1 = 'fungible'
>>> word2 = 'ephemeral'
>>> word3 = 'fragile'
>>> longer(word1, word2)
False
>>> longer(word2, word3)
True
```

Part C — Conditional Statements

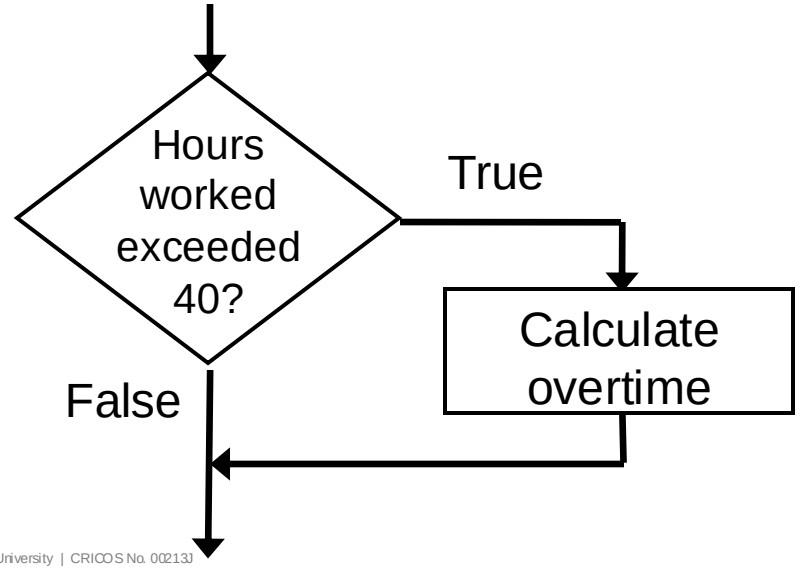
Key programming concept: Conditional statements

- So far we have seen how to execute fixed *sequences* of assignments and function calls
- However, a sequence always performs the *same* actions in the *same* order
- To give the computer a choice of actions we need to introduce other *control structures*

- There is one basic *conditional statement* in Python which can be used in a variety of ways, including:
  - A single-alternative `if` statement
  - A two-alternative `if-else` statement
  - A many-alternative `if-elif-else` statement
- In all cases we use Boolean expressions to guide the choice

Conditional actions

- Consider a program which calculates a person's pay
  - If they are eligible for overtime after working 40 hours, we need to calculate those extra hours at a higher rate of pay




```
# Step 1: Calculate normal hourly rate
normal_pay = hours_worked * pay_rate
overtime_pay = 0

# Step 2: Optionally calculate overtime pay
if hours_worked > 40:
    # Calculate overtime payment for hours over 40
    # (we only do this if overtime was worked)
    overtime_hours = hours_worked - 40
    overtime_pay = overtime_hours * pay_rate * 0.5

# Step 3: Sum total payment
pay = normal_pay + overtime_pay
```

Choosing between several actions

- An `if-elif-else` statement allows more than one condition to be checked
  - As soon as a condition evaluates to `True`, the corresponding statement block is executed
  - If all of the conditions evaluate to `False`, the statement block in the `else` clause (if any) is executed
  - Flow of control jumps to the end of the `if-elif-else` statement after one of its blocks is executed

```
if speed < 5: # km/hr
    print("Hurry up!")
elif speed < 50:
    print("Go a bit quicker!")
elif speed < 90:
    print("That's fast enough!")
else:
    print("Slow down!")

print("Give me the wheel!")
```

Example of choosing between several actions

```
if speed < 5: # km/hr
    print("Hurry up!")
elif speed < 50:
    print("Go a bit quicker!")
elif speed < 90:
    print("That's fast enough!")
else:
    print("Slow down!")

print("Give me the wheel!")
```

If this condition is true …

… then this message is printed …

… followed by this one

Example of choosing between several actions

```
if speed < 5: # km/hr
    print("Hurry up!")
elif speed < 50:
    print("Go a bit quicker!")
elif speed < 90:
    print("That's fast enough!")
else:
    print("Slow down!")

print("Give me the wheel!")
```

If this condition is false …

… and this condition is true …

… then this message is printed …

… followed by this one

Example of choosing between several actions

```
if speed < 5: # km/hr
    print("Hurry up!")
elif speed < 50:
    print("Go a bit quicker!")
elif speed < 90:
    print("That's fast enough!")
else:
    print("Slow down!")

print("Give me the wheel!")
```

If this condition is false …

… and this condition is false …

… and this condition is true …

… then this message is printed …

… followed by this one

Example of choosing between several actions

```
if speed < 5: # km/hr
    print("Hurry up!")
elif speed < 50:
    print("Go a bit quicker!")
elif speed < 90:
    print("That's fast enough!")
else:
    print("Slow down!")

print("Give me the wheel!")
```

If this condition is false …

… and this condition is false …

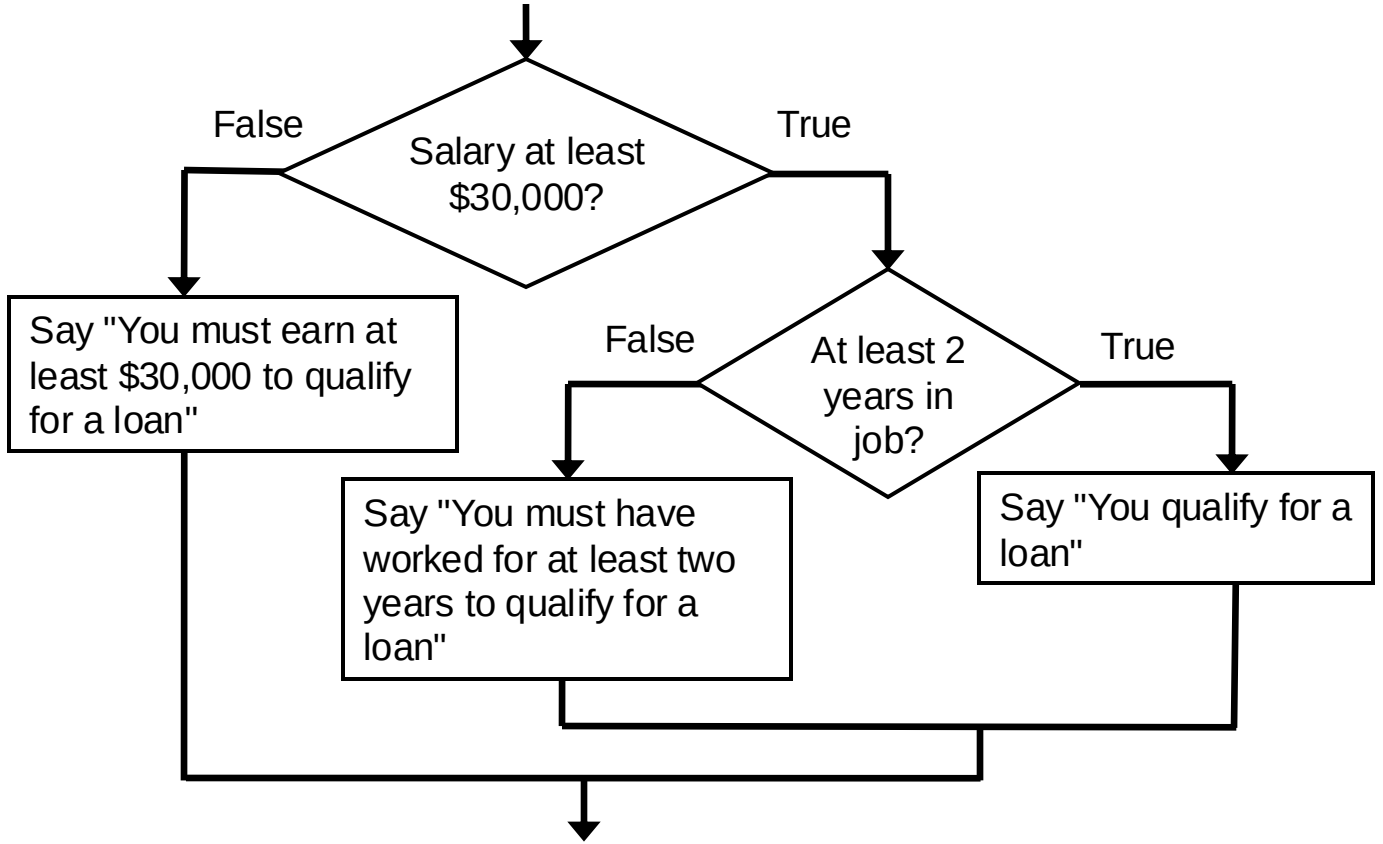
… and this condition is false …

… then this message is printed …

… followed by this one

Nested conditional statements

- Nested `if` statements allow us to build complex decision structures that will only test a particular condition if a previous condition has already evaluated to True
- As an example:
  - A bank has a rule that before a person qualifies for a loan, they must earn at least $30,000 and have worked in their current job for at least 2 years
  - Different rejection reasons are stated depending on the conditions not met



Example of nested conditional statements

```
if salary >= 30000:
    if years_in_job >= 2:
        print('You qualify for the loan')
    else:
        print('You must have worked in your',
              'current job for at least two',
              'years to qualify for a loan')
else:
    print('You must earn at least $30,000',
          'to qualify for a loan')
```

This entire `if` statement appears as one alternative of another

Before next week ...

1. If you haven't already done so, make sure you know how to locate the *Python Standard Library* reference manual
   - You will need to refer to it frequently from now on to find pre-defined functions and methods
   - You can download a PDF copy or access it online via IDLE's Help function
2. Review and try this week's demonstration files.
3. Complete this week's workshop exercises, before, during and after your workshop.
4. Make a start on Assessment Task 1!