

# Python Training

Kevin Cunningham

Grab repo at <https://github.com/doingandlearning/graphcore-python>

---

# A bit about me

- Lived, taught and developed in Brighton for 20 years
- Recently relocated back to Northern Ireland
- Dad to two boys
- Love learning new things
- You can find me on Twitter (@dolearning) or on my website (<https://kevincunningham.co.uk>)

# A bit about the course

Let's look at Mindnode

# Timings

---

9.30 - 10.45

Session 1

---

10.45 - 11

Coffee

---

12.00 - 1

Lunch

---

1.00 - 2.30

Session 3

---

2.30 - 2.45

Coffee

---

2.45 - 4.30

Session 4

---

# A bit about you

- Your name and role
- Your programming and Python experience
- What you're hoping to get out of this course

# Getting Started With Python

- Setting the scene
- Running Python script code
- Creating virtual environment

Demo folder: 01-GettingStarted

# 1. Setting the Scene

- Hello Python
- What can you do with Python?
- Downloading Python
- Using the Python documentation
- Installing Python packages

# Hello Python

- Python is a powerful and expressive programming language
  - Object-oriented
  - Dynamic typing
  - Interpreted
- Available on a wide range of platforms
  - Unix/Linux
  - Windows
  - Mac OS X
  - etc ...

# What can you do with Python?

- Scripting
- File I/O
- String handling and regular expressions
- Web applications and REST web services
- Data science

# Downloading Python

Are we all there?

- Anaconda
- PyCharm CE

# Using Python Documentation

- Docs available online at <https://docs.python.org>
-

# Installing Python Packages

There are many Python packages available

- e.g. NumPy, Matplotlib, etc.
- See <https://pypi.python.org/pypi> for details

You can use the pip package manager to install Python packages:

- For example, to install the NumPy package:

```
1 pip install numpy
```

To find where pip installed a package:

```
1 pip show numpy
```

Note: These are already installed in Anaconda

## 2. Running Python Script

- Running Python script interactively
- Creating variables
- Line continuation
- Blocks
- Creating and running Python modules
- Python keywords

# Running Python Script Interactively

Run the Python interpreter in interactive mode, and execute Python code.

```
1 python
```

Then enter some Python code. For example:

```
1 print("Hello World")
```

# Creating Variables

In Python, you don't need to declare a variable

- Just assign it a value, and Python will create it for you dynamically

Rules for identifiers in Python

- Can contain uppercase or lowercase letters, digits, and underscore
- But can't start with a digit

```
1  firstname = "Homer"
2  lastname = "Simpson"
3  fullname = firstname + " " + lastname
4  print(fullname)
```

# Line Continuation

If a statement spans multiple lines...

- You can use `\\` to continue from one line to the next

```
1  firstname = "Homer"
2  lastname = "Simpson"
3  fullname = firstname + \
4      " " + \
5  lastname
6  print(fullname)
```

# Blocks

Python uses indentation to denote blocks

- Don't use {}
- Use `:` to indicate the start of an indented block

```
1 age = 21
2 if age ≥ 18 and age ≤ 30:
3     print("You are eligible for an 18-30s holiday!")
4 print("That's all folks")
```

# Creating and Running Python Modules

You can put Python code into modules

- A module is just a script file containing Python code
- Typically starts with a lowercase letter and ends in `.py`

greeting.py

```
1 print("Hello Python!")
2 print("This is my module")
```

You can run the module via the Python interpreter

```
1 python greeting.py
```

# Python Keywords

Here is a full list of all the keywords in Python

- False, None, True
- if, elif, else, assert, is
- and, or, not
- for, in, from, while, break, continue, pass
- def, return, global, nonlocal, lambda
- import, from
- class, del
- raise, try, except, as, finally
- with, as
- yield
- await, async

Any questions?

# Annex: Creating a Virtual Environment

- Overview
- Installing virtualenv
- Creating a virtual environment for a project
- Activating a virtual environment
- Using virtualenv
- Deactivating a virtual environment

# Overview

In your life as a Python developer, you'll likely create many applications that use diverse Python packages

Ideally you would like the applications to be independent of each other

- The Python packages you download for one application shouldn't interfere with the Python packages for other applications

To help you keep Python application environments isolated from each other, you can use the `virtualenv` tool

# Installing virtualenv

You install virtualenv via pip as a one-off exercise as follows:

```
1 pip install virtualenv
```

You can test your installation as follows:

```
1 virtualenv --version
```

# Creating a virtual environment for a project

To create a virtual environment for a particular project:

```
1 virtualenv MyProject
```

This command creates a folder named MyProject that contains:

- Python executable files
- A copy of the pip library, which you can use to install other packages (locally for this virtual environment)

# Activating a virtual environment

To begin using a virtual environment, you must activate it

In Mac/Unix

```
1 source MyProject/bin/activate
```

In Windows:

```
1 MyProject\Scripts\activate
```

After you've activated a virtual environment, its name will appear in the command prompt.

# Using virtualenv

You can now use pip to install packages into your virtual environment.

- e.g. to install the 'request' packages:

```
1 pip install requests
```

This will install the package into Lib/site-packages folder.

You can write a Python script that uses the package

```
1 import requests
2 response = requests.get('https://httpbin.org/ip')
3 print('Your IP is {}'.format(response.json()['origin']))
```

And run it as normal

```
1 python main.py
```

# Deactivating a virtual environment

You can deactivate a virtual environment as follows:

```
1 deactivate
```

This tears down your virtual environment

- You don't see the packages in that virtual environment any more
- You can reactivate it whenever you need to (see 2 slides previous)

# Python Language Fundamentals

- Defining and using modules
- Defining and using packages
- Basic data types

# Defining and using modules

- The Python standard library
- Understanding modules
- More about modules
- Listing the names in a module

# The Python standard library

Python defines an extensive and powerful standard library

- Comprises a large number of modules

Built-in modules are implemented in C

- Provide access to low-level system functionality
- E.g. file I/O

Other modules are implemented in Python

- See the Lib folder in the Python installation folder

For full info, see: <https://docs.python.org/3.10/library>

# Understanding modules

You can create your own Python modules

- Here's a simple module, which just defines some variables

greetings.py

```
1 morning = "Good morning"
2 afternoon = "Good afternoon"
3 evening = "Good evening"
```

To use a module elsewhere, use the import keyword

- Several ways to do this:

```
1 import greetings
2 print(greetings.morning)
```

```
1 from greetings import morning, afternoon
2 print(morning + " " + afternoon)
```

```
1 from greetings import *
2 print(morning + " " + afternoon + " " + evening)
```

# More about modules

You can access the name of a module Use the **name** property

usegreetings

```
1 import greetings
2
3 print("Name of current module is %s" % __name__)
4 print("Name of greetings module is %s" % greetings.__name__)
```

Python only imports a given module once

- Regardless of how many times you try to import it

Python searches the following locations for a module

- The directory containing the input script (or the current directory)
- The directory specified by PYTHONPATH
- The installation-dependent default

# Listing the names in a module

You can list all the names defined in a module

- Use the `dir()` built-in function

listmodulenames.py

```
1 import math
2 from greetings import morning, afternoon
3
4 print("Names in the math module:")
5 print(dir(math))
6
7 print("\nNames in the current module:")
8 print(dir())
```

# Defining and using packages

- Overview of packages
- Example modules
- Importing specific modules
- Aliasing imported modules
- Importing all modules

# Overview of packages

Python allows you to organise related modules into packages and sub-packages

- A package is a folder that contains a file named `\_\_init\_\_.py`

## Example

```
1  utils/          Top-level package, named utils.  
2      __init__.py  
3      constants/    Sub-package for constants.  
4          __init__.py  
5          metric.py  
6          physics.py  
7          ...  
8      messages/     Sub-package for messages.  
9          __init__.py  
10         french.py  
11         norwegian.py  
12         ...
```

# Example modules

Here are the modules we've defined in the `utils` package. Modules in the `utils.constants` sub-package:

```
1 INCH_TO_CM = 2.54
2 MILE_TO_KM = 1.61
```

```
1 ELECTRONIC_CHARGE = 1.602e-19
2 PLANCKS_CONSTANT = 6.626e-34
```

Modules in `utils.messages` sub-package:

```
1 HELLO = "Bonjour"
2 GOODBYE = "Au revoir"
```

```
1 HELLO = "Hei"
2 GOODBYE = "Ha det bra"
```

# Importing specific modules

To import specific module(s) from a package:

```
1 import utils.constants.metric  
2  
3 print("Inch to centimetre: %.4f" % utils.constants.metric.INCH_TO_CM)  
4 print("Mile to kilometre: %.4f" % utils.constants.metric.MILE_TO_KM)
```

To import specific module(s) from a package, into the current symbol namespace:

```
1 from utils.constants import metric  
2  
3 print("Inch to centimetre: %.4f" % metric.INCH_TO_CM)  
4 print("Mile to kilometre: %.4f" % metric.MILE_TO_KM)
```

To import specific name(s) from a module from a package, into the current symbol namespace:

```
1 from utils.constants.metric import INCH_TO_CM, MILE_TO_KM  
2  
3 print("Inch to centimetre: %.4f" % INCH_TO_CM)  
4 print("Mile to kilometre: %.4f" % MILE_TO_KM)
```

# Aliasing imported modules

You can specify a local alias for a module

- Use `import ... as`

```
1 # import a module and give it an alias.
2 import utils.constants.metric as metric
3
4 print("Alias example")
5 print("Inch to centimetre: %.4f" % metric.INCH_TO_CM)
6 print("Mile to kilometre: %.4f" % metric.MILE_TO_KM)
```

# Importing all modules

You can use `*` to indicate you want to import all modules from a package

```
1  from utils.messages import *
2
3  print("Hello in French: %s" % utils.messages.french.HELLO)
4  print("Goodbye in French: %s" % utils.messages.french.GOODBYE)
5  print("Hello in Norwegian: %s" % utils.messages.norwegian.HELLO)
6  print("Goodbye in Norwegian: %s" % utils.messages.norwegian.GOODBYE)
```

You must tell Python which modules to actually import from that package

- In the package's `init.py` file ...
- Define a global variable named `all` and set it to a list of all the modules to be imported

```
1  __all__ = ["french", "norwegian"]
```

# Basic data types

- Numbers
- Numeric operators
- Bitwise operators
- Using the math module
- Booleans
- Relational operators
- Boolean logic operators
- Operator precedence
- Strings
- Other built-in types

# Numbers

Python has three numeric types

- Integers
- Floating point numbers
- Complex numbers

```
1  i1 = 12345
2  i2 = 1234567890123456789
3  i3 = int("123", 8)
4  print("%d %d %d" % (i1, i2, i3))
5
6  f1 = 1.23
7  f2 = 4.56e-34
8  f3 = 7.89e+34
9  f4 = float("123.45")
10 print("%g %g %g %g" % (f1, f2, f3, f4))
11
12 c1 = 1 + 2j
13 c2 = 3 - 4j
14 c3 = 5j
15 c4 = complex("6+7j")
16 print("%g + %gi" % (c1.real, c1.imag))
17 print("%g + %gi" % (c2.real, c2.imag))
18 print("%g + %gi" % (c3.real, c3.imag))
19 print("%g + %gi" % (c4.real, c4.imag))
```

# Numeric operators

Python supports the following operators on numbers

- $x^{**} y$
- `pow(x, y)`
- `divmod(x, y)`
- `c.conjugate()`
- `complex(re, im)`
- `float(x)`
- `int(x)`
- `abs(x)`
- $+x$
- $-x$
- $x \% y$
- $x // y$
- $x / y$
- $x ^ y$
- $x - y$
- $x + y$

# Using the math module

The math module defines several useful mathematical constants and functions For details, see  
<https://docs.python.org/3.10/library/math.html>

## Example

```
1 import math
2
3 print(dir(math))
4
5 print("pi is %f" % math.pi)
6 print("360 degrees in radians is %g" % math.radians(360))
7 print("2 * pi radians in degrees is %g" % math.degrees(2 * math.pi))
8
9 print("sin(90 degrees) is %.4f" % math.sin(math.pi / 2))
10 print("cos(90 degrees) is %.4f" % math.cos(math.pi / 2))
11 print("acos(0) is %g degrees" % math.degrees(math.acos(0)))
12
13 print("hypoteneuse of right-angled triangle (sides 3, 4) is %g" % math.hypot(3, 4))
14 print("5 factorial is %g" % math.factorial(5))
```

# Booleans

Boolean is a built-in type

- Represents truth or falsehood

The following values are considered false:

- None
- False
- Zero of any numeric type, e.g. 0, 0.0, 0j
- Any empty sequence, e.g. "", (), []
- Any empty mapping, e.g. {}

All other values are considered true

- Including the True keyword ☺

# Relational operators

Python supports the following relational operators

- <
- <=
- >
- >=
- ==
- !=
- is
- is not

# Boolean logic operators

Python has three boolean logic operators:

- not
- and
- or

Example

```
1 month = int(input("Enter a month number [1-12]: "))
2
3 is_summer = month ≥ 6 and month ≤ 8
4 is_winter = month == 12 or month == 1 or month == 2
5 is_transition_season = not(is_winter or is_summer)
6
7 print("%s %s %s" % (is_summer, is_winter, is_transition_season))
```

# Operator precedence

This table shows the precedence of all operators from highest to lowest

---

# Strings

A string is an immutable sequence of Unicode characters

Can enclose in single quotes, double quotes, or triple quotes

```
1 str1 = "The computer says 'No' I'm afraid."
2 str2 = '<a href="www.bbc.co.uk">Click here for the BBC</a>'
3
4 str3 = """Birthday present ideas:
5     - Bugatti Chiron
6     - 4xHD OLED 64-inch TV
7     - Socks"""
8
9 print("%s\n%s\n%s" % (str1, str2, str3))
```

The String class defines many methods For details, see <https://docs.python.org/3.10/library/string.html>

There's also excellent support for regular expressions

For details, see <https://docs.python.org/3.10/library/re.html>

# Other built-in types

## Text sequence types

- String - see previous slide

## Basic sequence types

- List, tuple, and range

## Binary sequence types

- bytes, bytearray, and memoryview

## Set types

- set, frozenset

## Mapping type

- dict

Any questions?

# Control Flow

- Conditional Statements
- Loops

# Conditional Statements

- Using if tests
- Nesting if tests
- Using the if-else operator
- Doing nothing
- Testing a value is in a set of values
- Testing a value is in a range

# Using if tests

## Basic if tests

```
1 if expression:  
2   body
```

## if-else tests

```
1 if expression:  
2   body1  
3 else:  
4   body2
```

## if-elif tests

```
1 if expression1 :  
2   body1  
3  
4 elif expression2 :  
5   body2  
6  
7 elif expression3 :  
8   body3  
9 ...  
10 else :  
11   lastBody
```

## Notes:

- Test conditions can be any type of expression
- Use indentation to indicate the extent of a block,  
i.e. don't use {}

# Nesting if tests

You can nest if tests inside each other

- Use indentation to indicate level of nesting

Example:

```
1  age = int(input("Please enter your age: "))
2  gender = input("Please enter your gender [M/F]: ").lower()
3
4  if age < 18:
5      if gender == "m":
6          print("Boy")
7      else:
8          print("Girl")
9
10     else:
11         if age >= 100:
12             print("Centurion")
13
14         if gender == "m":
15             print("Man")
16         else:
17             print("Woman")
18
19     print("The End")
```

# Using the if-else operator

The if-else operator is an in-situ test

- trueResult if condition else falseResult

Example:

```
1  isMale = ...
2  age    = ...
3
4  togo = (65 - age) if isMale else (60 - age)
5
6  print("%d years to retirement" % togo)
```

# Doing nothing

If you're not sure what to do if a test is true...

- You can use the `pass` statement
- Equivalent to a `null` statement in other languages

Example:

```
1 team = input("Who is your favourite rugby team? ")
2
3 if team == "Ireland":
4     pass      # Eeek. We'll need to do something about this!
5
6 print("Your favourite team is %s" % team)
```

# Testing a value is in a set of values

You can test if a value is in a set of allowable values

- Use the in operator

Example:

```
1 country = input("Please enter your country: ")
2
3 if country in ("Netherlands", "Belgium", "Luxembourg"):
4     print("Lowlands country")
5
6 elif country in ("Norway", "Sweden", "Denmark", "Finland", "Iceland"):
7     print("Nordic country")
8
9 elif country in ("England", "Scotland", "Wales", "Northern Ireland"):
10    print("UK country")
11
12 else:
13     print("%s isn't classified in this particular application!" % country)
```

# Testing a value is in a range

You can test if a value is in a range of allowable values

- Call `range(start,end)` to return a range
- The range is inclusive at start, exclusive at the end

Example:

```
1  number = int(input("Enter a football jersey number [1 to 11]: "))
2
3  if number == 1:
4      print("Goalie")
5
6  elif number in range(2, 6):
7      print("Defender")
8
9  elif number in range(6, 10):
10     print("Midfielder")
11
12 else:
13     print("Striker")
```

# Loops

- Using while loops
- Using for loops
- Using for loops with a range
- Unconditional jumps
- Using else in a loop
- Simulating do-while loops

# Using while loops

The while loop is the most straightforward loop construct

```
1 while expression:  
2   loopBody
```

- Test expression is evaluated
- If true, loop body is executed
- Test expression is re-evaluated
- Etc...

Note:

- Loop body will not be executed if test is false initially

Example:

```
1 print("Numbers from 1-5 inclusive")  
2 i = 1  
3 while i <= 5:  
4   print(i)  
5   i = i + 1
```

# Using for loops

The for loop is different than in most languages

- In Python, a for loop iterates over items in a sequence

```
1 for item in sequence:  
2     loopBody
```

Example:

```
1 lottonumbers = [2, 7, 3, 12, 19, 1]  
2  
3 for item in lottonumbers:  
4     print(item)
```

# Using for loops with a range

You can also use a for loop to iterate over a numeric range

- Use range() to create a range of numbers
- The for loop will iterate over these numbers

Example:

```
1  print("Numbers from 0-4 inclusive")
2  for i in range(5):
3      print(i)
4
5  print("Numbers from 6-10 inclusive")
6  for i in range(6, 11):
7      print(i)
8
9  print("First 5 odd numbers")
10 for i in range(0, 9, 2):
11     print(i + 1)
```

# Unconditional jumps

Python provides two ways to perform an unconditional jump in a loop

- `break`
- `continue`

Example:

```
1  magicnumber = int(input("What is the magic number? "))
2
3  print("This loop terminates if it hits the magic number")
4  for i in range(1, 21):
5      if i == magicnumber:
6          break
7      print(i)
8  print("End")
9
10 print("\nThis loop skips the magic number")
11 for i in range(1, 21):
12     if i == magicnumber:
13         continue
14     print(i)
15 print("End")
```

# Using else in a loop

You can define an else clause at the end of a loop

- Same kind of syntax as if...else
- The else branch is executed if the loop terminates naturally (i.e. if it didn't exit because of a break)

Example

```
1 magicnumber = int(input("What is the magic number? "))
2
3 print("This loop does some processing if it doesn't detect the magic number")
4 for i in range(1, 21):
5     if i == magicnumber:
6         break
7     print(i)
8 else:
9     print("The magic number %d was not detected" % magicnumber)
10
11 print("End")
```

# Simulating do-while loops

Many languages have a do-while loop

- Guarantees at least one iteration through the loop body
- The test is at the end, to determine whether to repeat

Python doesn't have a do-while loop, but you can emulate it as follows

```
1 while True:  
2     exammark = int(input("Enter a valid exam mark: "))  
3     if exammark ≥ 0 and exammark ≤ 100:  
4         break  
5  
6 print("Your exam mark is %d" % exammark)
```

Any questions?

# Functions

- Getting started with functions
- Going further with functions

# Getting started with functions

- Simple functions
- Passing arguments to a function
- Returning a value from a function
- Understanding scope

# Simple functions (1/2)

A function is a named block of code

- Starts with the `def` keyword
- Followed by the name of the function
- Followed by parentheses, where you can define arguments
- Followed by a block, where you define the function body

```
1 def name_of_function(arg1, arg2, ..., argn):  
2     statements  
3     statements  
4     ...
```

To call a function

- Specify the function name
- Followed by parentheses, where you can pass arguments

```
1 name_of_function(argvalue1, argvalue2, ..., argvaluen)
```

# Simple functions (2/2)

Here's an example of how to define and call simple functions

```
1 def say_goodmorning():
2     print("Start of say_goodmorning")
3     print(" Good morning!")
4     print("End of say_goodmorning\n")
5
6 def say_goodafternoon():
7     print("Start of say_goodafternoon")
8     print(" Good afternoon!")
9     print("End of say_goodafternoon\n")
10
11 def say_goodevening():
12     pass
13
14
15 # Usage (i.e. client code)
16 say_goodmorning()
17 say_goodafternoon()
18 say_goodevening()
19
20 f = say_goodmorning
21 f()                                     # Calls say_goodmorning() really
22
23 print("THE END")
```

# Passing arguments to a function

You can pass arguments to a function

- In the function definition, declare the argument names in the parentheses
- In the client code, pass argument values in the call

Example

```
1 def display_message(message, count):  
2     for i in range(count):  
3         print(message)  
4  
5 # Usage (i.e. client code)  
6 display_message("Hello", 3)  
7 display_message("Goodbye", 1)
```

# Returning a value from a function

Functions can return a value, via a return statement

- If you don't return a value explicitly, the function returns None

Example:

```
1 def display_message(msg):
2     print(msg)
3
4 def generate_hyperlink(href, text):
5     return "<a href='{0}'>{1}</a>".format(href, text)
6
7 def get_number_in_range(msg, lower, upper):
8     while True:
9         num = int(input(msg))
10        if num ≥ lower and num < upper:
11            return num
12
13
14 # Usage (i.e. client code)
15 result1 = display_message("Hello world")
16 print("result1 is %s" % result1)
17
18 result2 = generate_hyperlink("http://www.bbc.co.uk", "Favourite month? ")
19 print("result2 is %s" % result2)
20
21 result3 = get_number_in_range("Favourite month? ", 1,
22 print("result3 is %s" % result3)
```

# Understanding scope (1/2)

If you declare a variable outside a function:

- The variable is global to the module
- Prefix the name with `_` to make it private to this module

If you declare a variable inside a function:

- The variable is local to the function

If you want to assign a global variable inside a function:

- You must declare the variable inside the function, using the `global` keyword
- Tells the Python interpreter it's an existing global name, not a new local name

# Understanding scope (2/2)

This example shows how to define and use global variables

```
1  __DBNAME = None
2
3  def initDB(name):
4      global __DBNAME
5      if __DBNAME is None:
6          __DBNAME = name
7      else:
8          raise RuntimeError("Database name has already been set.")
9
10 def queryDB():
11     print("TODO, add code to query %s" % __DBNAME)
12
13 def updateDB():
14     print("TODO, add code to update %s" % __DBNAME)
15
16
17 # Usage (i.e. client code)
18 initDB("Server=.;Database=Northwind")
19 queryDB()
20 updateDB()
```

# Going further with functions

- Default argument values
- Variadic functions
- Passing keyword arguments
- Variadic keyword arguments
- Built-in functions
- Examples of using functions

# Default argument values

You can define default argument values for a function

- In the function definition, specify default values as appropriate
- In the client code, pass argument values or rely on defaults

Example:

```
1 def book_flight(fromairport, toairport, numadults=1, numchildren=0):
2     print("\nFlight booked from %s to %s" % (fromairport, toairport))
3     print("Number of adults: %d" % numadults)
4     print("Number of children: %d" % numchildren)
5
6 # Usage (i.e. client code)
7 book_flight("BRS", "VER", 2, 2)
8 book_flight("LHR", "VIE", 4)
9 book_flight("LHR", "OSL")
```

# Variadic functions

Python allows you to define a function that can take any number of arguments

- In the function definition, prefix the last argument name with \*
- Internally, these arguments will be wrapped up as a tuple
- You can iterate through the tuple items by using a for loop

Example

```
1 def display_favourite_things(name, *things):
2     print("Favourite things for %s" % name)
3     for item in things:
4         print("  %s" % item)
5
6 # Usage (i.e. client code)
7 display_favourite_things("Kath", "Ethan", "Caleb", 3, "Reading", "Learning", "Climbing")
```

# Passing keyword arguments

Client code can pass arguments by name

- Use the syntax argument\_name = value

Useful if the function has a lot of default argument values

- Client code can choose exactly which arguments to pass in

Example:

```
1 def book_flight(fromairport, toairport, numadults=1, numchildren=0):  
2     print("\nFlight booked from %s to %s" % (fromairport, toairport))  
3     print("Number of adults: %d" % numadults)  
4     print("Number of children: %d" % numchildren)  
5  
6 # Usage (i.e. client code)  
7 book_flight("BRS", "VER", 2, 2)  
8 book_flight("LHR", "CDG", numchildren=2)  
9 book_flight(numchildren=3, fromairport="LGW", toairport="NCE")
```

# Variadic keyword arguments

It's also possible to define variadic keyword arguments

- Use `**` rather than `*` on the argument
- Allows you to pass in any number of keyword args

Internally, the arguments are wrapped as a dictionary

- You can iterate through the key/value pairs by using a for loop

Example

```
1 def myfunc(**kwargs):
2     for k, v in kwargs.items():
3         print ("key %s, value %s" % (k, v))
4
5 # Usage (i.e. client code)
6 myfunc(favTeam="Ireland", favNum=3, favColour="green")
```

# Built-in functions

Python has a suite of built-in functions that are always available

---

# Examples of Using Function (1/2)

I've written some examples to illustrate how to use functions in realistic scenarios

- Processing lines of text from a file
- Using regular expressions to find particular values in the file

Demo location Demos\04-Functions\WorkedExamples

# Examples of using functions (2/2)

To open and read a file:

- Call open() to open a file - returns a file handle
- To read lines from the file, simply iterate over the file handle

To use regular expressions:

- The re module has compile() and search() functions to compile and use a regular expression

Here's the first example:

```
1 import re
2
3 pattern = re.compile('Attribute ID \(\0xC2\)')
4
5 with open('data.txt') as fh:
6     for line in fh:
7         result = pattern.search(line)
8         if result:
9             print(line)
```

Any questions?

# Data Structures

- Sequence types
- Using sequences
- Set types
- Mapping types
- Additional techniques
- Worked examples

# 1. Sequence Types

- Overview
- Lists
- Splitting and joining
- Tuples
- Ranges

# Overview

.

## Basic sequence types

- List, tuple, and range

## Text sequence types

- String

## Binary sequence types

- bytes, bytearray, and memoryview

# Lists

There are several ways to create a list

- []
- [item, item, item ... ]
- list()
- list(iterable)

Example:

```
1  list1 = []
2  list2 = ["Italy", "France", "Spain"]
3  list3 = [3, 12, 19, 1, 2, 7]
4  list4 = list()
5  list5 = list(list3)
6  list6 = list("Hello")
7
8  print("list1 has %d items: %s" % (len(list1), list1))
9  print("list2 has %d items: %s" % (len(list2), list2))
10 print("list3 has %d items: %s" % (len(list3), list3))
11 print("list4 has %d items: %s" % (len(list4), list4))
12 print("list5 has %d items: %s" % (len(list5), list5))
13 print("list6 has %d items: %s" % (len(list6), list6))
```

# Splitting and Joining

A common scenario where lists crop up in Python is when you call `split()` or `join()` on a string

- `split()` - splits a string into a list of substrings
- `join()` - joins a list into a concatenated string

Example:

```
1 str = "and we were singing, hymns and arias, land of my fathers, ar hyd yr nos"
2
3 words = str.split(", ")
4
5 lines = "... \n".join(words)
6
7 print("%s" % lines)
```

# Tuples

There are several ways to create a tuple

- ()
- a, or (a,)
- a,b,c or (a,b,c)
- tuple()
- tuple(iterable)

Example:

```
1  tuple1 = ()  
2  tuple2 = "Norway",      # or: tuple2 = ("Norway",)  
3  tuple3 = 3, 19, 2      # or: tuple3 = (3, 19, 2)  
4  tuple4 = tuple()  
5  tuple5 = tuple(tuple3)  
6  
7  print("tuple1 has %d items: %s" % (len(tuple1), tuple1))  
8  print("tuple2 has %d item(s): %s" % (len(tuple2), tuple2))  
9  print("tuple3 has %d items: %s" % (len(tuple3), tuple3))  
10 print("tuple4 has %d items: %s" % (len(tuple4), tuple4))  
11 print("tuple5 has %d items: %s" % (len(tuple5), tuple5))
```

# Ranges

To create a range, use the range constructor

- `range(stop)`
- `range(start, stop)`
- `range(start, stop, step)`

Example:

```
1  def display_range(msg, r):
2      print("\n" + msg)
3      for i in r:
4          print(i)
5
6  range1 = range(5)
7  range2 = range(5,10)
8  range3 = range(5,10,2)
9
10 display_range("range1", range1)
11 display_range("range2", range2)
12 display_range("range3", range3)
```

## 2. Using Sequences

- Common sequence operations
- Slicing operations
- Unpacking operations
- Sequence modification operations
- Optional exercise

# Common sequence operations

You can perform these operations on any sequence:

```
1 euro = ["GB", "ES", "NL", "F", "D", "I", "P"]
2 asia = ["SG", "JP"]
3
4 print("%s" % "P" in euro)                      # True
5 print("%s" % "F" not in euro)                    # False
6 print("%s" % (euro + asia))                     # ['GB', 'ES', 'NL', 'F', 'D', 'I', 'P', 'SG', 'JP']
7 print("%s" % (asia * 2))                        # ['SG', 'JP', 'SG', 'JP']
8 print("%s" % (2 * asia))                        # ['SG', 'JP', 'SG', 'JP']
9 print("%d" % len(euro))                         # 7
10 print("%s" % min(euro))                        # D
11 print("%s" % max(euro))                        # P
12 print("%d" % euro.index("NL"))                 # 2
13 print("%d" % euro.index("NL", 1))              # 2
14 print("%d" % euro.index("NL", 1, 4))           # 2
15 print("%d" % euro.count("ES"))                 # 1
```

# Slicing operations

```
1 euro = ["GB", "ES", "NL", "F", "D", "I", "P"]
2 asia = ["SG", "JP"]
3
4 print("%s" % (euro[1]))                      # ES
5 print("%s" % (euro[1:5]))                     # ['ES', 'NL', 'F', 'D']
6 print("%s" % (euro[1:5:2]))                   # ['ES', 'F']
7 print("%s" % (euro[3:]))                       # ['F', 'D', 'I', 'P']
8 print("%s" % (euro[: -3]))                    # ['GB', 'ES', 'NL', 'F']
```

# Unpacking operations

You can unpack (i.e. extract) elements in a sequence

The following example illustrates the techniques available

```
1 euro = ["GB", "ES", "NL", "F"]
2
3 # Manually getting items.
4 a, b, c, d = euro[0], euro[1], euro[2], euro[3]
5 print("%s %s %s %s" % (a, b, c, d))      # GB ES NL F
6
7 # Unpacking.
8 e, f, g, h = euro
9 print("%s %s %s %s" % (e, f, g, h))      # GB ES NL F
10
11 # Catch-all unpacking.
12 i, j, *k = euro
13 print("%s %s %s" % (i, j, k))            # GB ES ['NL', 'F']
```

# Sequence modification operations

You can perform these operations on a mutable sequence such as a list:

```
1 euro = ["GB", "ES", "NL", "F"]
2
3 euro[0] = "CY"
4 euro[1:3] = ["US", "AU", "AT"]
5 euro.append("SW")
6 euro.extend(["YU", "ZR"])
7 euro.insert(1, "NI")
8 print("%s" % euro)      # ['CY', 'NI', 'US', 'AU', 'AT', 'F', 'SW', 'YU', 'ZR']
9
10 euro.pop()
11 euro.pop(1)
12 del euro[2:4]
13 print("%s" % euro)      # ['CY', 'US', 'F', 'SW', 'YU']
14
15 euro.remove("US")
16 euro.reverse()
17 print("%s" % euro)      # ['YU', 'SW', 'F', 'CY']
18
19 eurocopy = euro.copy()
20 euro.clear()
21 print("%s" % eurocopy)  # ['YU', 'SW', 'F', 'CY']
22 print("%s" % euro)      # []
```

# Exercise

Write a Python program as follows:

- Ask the user to enter a series of numbers (-1 to quit)
- Determine which numbers are prime
- Display the prime numbers on the console

For the solution code: See Solutions\05-DataStructures\primes.py

Here are more detailed instructions for this exercise:

1. Write a function named `get_numbers()`. The function should loop around, asking the user to enter a number. For each number, add it to a list. Stop looping when the user enters -1. Return the list at the end of the function.
2. Write a function named `find_primes()`. The function takes one argument - a list of numbers. The function should loop through the numbers, to find the prime numbers. The function should return the prime numbers.
3. Write a function named `display_numbers()`. The function takes one argument - a list of numbers. The function displays the numbers on the screen.

Call these functions from your "main" code, to get numbers from the user, find which ones are prime,

# 3. Set Types

- Creating a set
- Creating a frozen set
- Common set operations
- Set modification operations

# Creating a set

There are several ways to create a set

- {item, item, item, ... }
- set()
- set(iterable)
- Via a comprehension, similar to lists

Example:

```
1  set1 = {"dog", "ant", "bat", "cat", "dog"}  
2  set2 = set()  
3  set3 = set(("dog", "ant", "bat", "cat", "dog"))  
4  set4 = set("abracadabra")  
5  set5 = {c.upper() for c in "abracadabra"}  
6  
7  print("set1 has %d items: %s" % (len(set1), set1))  
8  print("set2 has %d items: %s" % (len(set2), set2))  
9  print("set3 has %d items: %s" % (len(set3), set3))  
10 print("set4 has %d items: %s" % (len(set4), set4))  
11 print("set5 has %d items: %s" % (len(set5), set5))
```

# Creating a frozen set

Creating a frozenset is similar to creating a set

- Use the frozenset constructor

Example:

```
1  set1 = frozenset({"dog", "ant", "bat", "cat", "dog"})
2  set2 = frozenset()
3  set3 = frozenset(("dog", "ant", "bat", "cat", "dog"))
4  set4 = frozenset("abracadabra")
5  set5 = frozenset({c.upper() for c in "abracadabra"})
6
7  print("set1 has %d items: %s" % (len(set1), set1))
8  print("set2 has %d items: %s" % (len(set2), set2))
9  print("set3 has %d items: %s" % (len(set3), set3))
10 print("set4 has %d items: %s" % (len(set4), set4))
11 print("set5 has %d items: %s" % (len(set5), set5))
```

# Common set operations

You can perform these operations on any set:

```
1  s1 = {"GB", "US", "SG"}  
2  s2 = {"GB", "US", "AU"}  
3  s3 = {"F", "BE", "CA"}  
4  
5  print("%s" % ("GB" in s1))      # True  
6  print("%s" % ("GB" not in s1))    # False  
7  print("%s" % (s1.isdisjoint(s2))) # False  
8  print("%s" % (s1.isdisjoint(s3))) # True  
9  print("%s" % (s1.issubset(s2)))   # False  
10 print("%s" % (s1 <= s2))        # False  
11 print("%s" % (s1 < s2))         # False  
12 print("%s" % (s1.issuperset(s2)))# False  
13 print("%s" % (s1 >= s2))       # False  
14 print("%s" % (s1 > s2))        # False  
15 print("%s" % (s1.union(s2, s3)))# {'GB', 'US', 'BE', 'F', 'CA', 'AU', 'SG'}  
16 print("%s" % (s1 | s2 | s3))    # {'GB', 'US', 'BE', 'F', 'CA', 'AU', 'SG'}  
17 print("%s" % (s1.difference(s2, s3)))# {'SG'}  
18 print("%s" % (s1 - s2 - s3))    # {'SG'}  
19 print("%s" % (s1.symmetric_difference(s2)))# {'AU', 'SG'}  
20 print("%s" % (s1 ^ s2))         # {'AU', 'SG'}
```

# Set modification operations

You can perform these operations on a mutable set:

```
1 s1.add("HK")
2 s1.remove("US")
3 s1.discard("D")
4 print("%s" % s1)      # {'SG', 'HK', 'GB'}
5
6 print("%s" % s1.pop()) # SG
7 print("%s" % s1)      # {'HK', 'GB'}
8
9 s1.update(s2,s3)
10 s1 |= s4 | s5
11 print("%s" % s1)      # {'D', 'AU', 'US', 'I', 'F', 'P', 'N', 'GB', 'CA', 'HK'}
12
13 s1.intersection_update(s2,s3)
14 s1 &= s4 & s5
15 print("%s" % s1)      # {'GB', 'US'}
16
17 s1.difference_update({"AA", "BB"}, {"CC", "GB"})
18 s1 -= {"DD", "EE"} | {"FF", "GG"}
19 print("%s" % s1)      # {'US'}
20
21 s1.symmetric_difference_update(s2)
22 s1 ^ s2
```

# 4. Mapping Types

- Creating a dictionary
- Iterating over a dictionary
- Accessing items in a dictionary

# Creating a dictionary

There are several ways to create a dict

- {key:value, key:value, ... }
- dict()
- dict(anotherDict)
- dict(keyword=value, keyword=value, ... )
- dict(zip(keysIterable, valuesIterable))

Example:

```
1  dict1 = {"us":"+1", "nl":"+31", "no":"+47"}  
2  dict2 = dict()  
3  dict3 = dict({"us":"+1", "nl":"+31", "no":"+47"})  
4  dict4 = dict(us="+1", nl="+31", no="+47")  
5  dict5 = dict(zip(["us", "nl", "no"], ["+1", "+31", "+47"]))  
6  
7  print("dict1 has %d items: %s" % (len(dict1), dict1))  
8  print("dict2 has %d items: %s" % (len(dict2), dict2))  
9  print("dict3 has %d items: %s" % (len(dict3), dict3))  
10 print("dict4 has %d items: %s" % (len(dict4), dict4))  
11 print("dict5 has %d items: %s" % (len(dict5), dict5))
```

# Iterating over a dictionary

There are several ways to iterate over a dict

- Iterate over the items (i.e. key-value pairs)
- Iterate over the keys
- Iterate over the values

Example:

```
1  dialcodes = {"us": "+1", "nl": "+31", "no": "+47"}  
2  
3  print("Items:")  
4  for k,v in dialcodes.items():  
5      print(k, v)  
6  
7  print("\nKeys:")  
8  for k in dialcodes.keys():  
9      print(k)  
10  
11 print("\nValues:")  
12 for v in dialcodes.values():  
13     print(v)
```

# Accessing items in a dictionary

There are various operations for accessing items in a dict

```
1  dialcodes = {"us": "+1", "nl": "+31", "no": "+47", "it": "+39"}  
2  
3  print("%s" % "us" in dialcodes)          # True  
4  print("%s" % "us" not in dialcodes)        # False  
5  
6  dialcodes["uk"] = "+44"  
7  print(dialcodes["uk"])                      # +44  
8  print(dialcodes.get("fr"))                  # None  
9  print(dialcodes.get("fr", "xxx"))           # xxx  
10  
11 del dialcodes["no"]  
12 print(dialcodes.pop("uk"))                  # +44  
13 print(dialcodes.pop("uk", "xxx"))           # xxx  
14 print(dialcodes.setdefault("it", "???"))    # ???  
15  
16 dialcodes.update({"ca": "+1", "it": "+39"})  
17 print(dialcodes) # {'ca': '+1', 'us': '+1', 'nl': '+31'}
```

# 5. Additional Techniques

- Generators
- List comprehensions
- Set comprehensions
- Dictionary comprehensions
- Filtering, sorting, and mapping
- Working with JSON data

# Generators

A generator is a special kind of function that returns a collection, one item at a time

- Use the `yield` keyword to yield the next value on each call

Example - consider the following two functions

- The 1st version returns a collection "all at once"
- The 2nd version yields a collection one element at a time

```
1 def getNums():
2     nums = []
3     while True:
4         num = int(input("Number? "))
5         if num == -1:
6             break
7         nums.append(num)
8     return nums
9
10 # Client code.
11 nums = getNums()
12 for n in nums:
13     print(" %d" % n)
```

```
1 def getNumsB():
2     while True:
3         num = int(input("Number? "))
4         if num == -1:
5             break
6         yield num
7
8
9
10 # Client code.
11 nums = getNums()
12 for n in nums:
13     print(" %d" % n)
```

# List comprehensions

You can create a list from another sequence

- Apply an operation on all the items in an existing sequence
- This is known as a "list comprehension"

Example:

```
1 squares = [x**2 for x in range(6)]
2
3 ftemps = [32, 68, 212]
4 ctemps = [(f-32)*5/9 for f in ftemps]
5
6 print("squares: %s" % squares)
7 print("ftemps: %s" % ftemps)
8 print("ctemps: %s" % ctemps)
```

# Set comprehensions

You can also create a "set comprehension"

- i.e. a set created from another sequence

Example:

```
1 ftemps = range(0, 50, 5)
2 ctemps = { int((f-32)*5/9) for f in ftemps }
3
4 print("ctemps: %s" % ctemps)
```

# Dictionary comprehensions

You can also create a "dictionary comprehension"

- i.e. a collection of key/value pairs created from another sequence

Example:

```
1 mydict = { i : i*i for i in range(1, 6) }
2
3 print("mydict: %s" % mydict)
```

# Filtering, sorting, and mapping (1/2)

Python defines functions that allow you to filter, sort, and map (i.e. transform) the elements in a collection

## Example

```
1 names = ["Zak", "Tim", "Ben", "Joe", "Kim", "Bud", "Te  
2  
3 bnames = list(filter(startsWithB, names))  
4 print(bnames)  
5  
6 sortedBnames = sorted(bnames)  
7 print(sortedBnames)  
8  
9 mappedSortedBnames = list(map(topAndTail, sortedBnames  
10 print(mappedSortedBnames)
```

```
1 def startsWithB(element):  
2     if len(element) and element[0] == 'B':  
3         return True  
4     else:  
5         return False
```

```
1 def topAndTail(element):  
2     return "***" + element + "***"
```

# Filtering, sorting, and mapping (2/2)

The sorted() function takes two optional arguments, which allow you to take control over the sorting

- key - function that indicates what aspect to sort items on
- reverse - boolean (default is false, i.e. ascending order)

## Example

```
1 names = ["Kevin", "Jayne", "Em", "Tom"]
2
3 sortedNamesAlphabetically = sorted(names)
4 print(sortedNamesAlphabetically)
5
6 sortedNamesByLength = sorted(names, key=personNameLength)
7 print(sortedNamesByLength)
8
9 sortedNamesByLengthDescending = sorted(names, key=personNameLength, reverse=True)
10 print(sortedNamesByLengthDescending)
```

# Working with JSON data (1/3)

JSON is a popular string data format

- Typically used for passing data to/from REST services
- Very easy to read/write JSON data in JavaScript (and in Python ☺)

Here are some example JSON strings:

```
1 personJson = '{ "name": "Kevin", "age": 21, "height": 1.67, "isWelsh": true }'  
2 coordsJson = '[ { "x": 100, "y": 150 }, { "x": 200, "y": 250 } ]'
```

To read/write JSON data in Python, use the standard Python module named `json`

- `json.loads()` loads JSON data into a Python dictionary/list
- `json.dumps()` dumps a Python dictionary/list into a JSON string

# Working with JSON data (2/3)

These examples show how to load JSON data into Python data structures

```
1 import json
2
3 personJson = '{"name": "Kevin", "age": 21, "height": 1.67, "isWelsh": true }'
4
5 person = json.loads(personJson)
6
7 print("%s is %d years old" % (person["name"], person["age"]))
8 print("Height is %.2f, Welshness is %s" % (person["height"], person["isWelsh"]))
```

```
1 import json
2
3 coordsJson = '[ { "x": 100, "y": 150 }, { "x": 200, "y": 250 } ]'
4
5 coords = json.loads(coordsJson)
6
7 print("Point 0 is %d, %d" % (coords[0]["x"], coords[0]["y"]))
8 print("Point 1 is %d, %d" % (coords[1]["x"], coords[1]["y"]))
```

Also see `readJsonFromFile.py` and `sampledata.json`

# Working with JSON data (3/3)

These examples show how to dump Python data into a JSON string

```
1 import json
2
3 person = {"name": "Kevin", "age": 21, "height": 1.67, "isWelsh": True }
4
5 personJson = json.dumps(person, indent=4)
6
7 print(personJson)
```

```
1 import json
2
3 coords = [ { "x": 100, "y": 150 }, { "x": 200, "y": 250 } ]
4
5 coordsJson = json.dumps(coords, indent=4)
6
7 print(coordsJson)
```

Any questions?

# Object Oriented Programming (OOP)

- Essential concepts
- Defining and using a class
- Class-wide members

# Essential concepts

- What is a class?
- What is an object?
- Class diagrams

# What is a class?

A class is a representation of a real-world entity

- Defines data, plus methods to work on that data
- You can hide data from external code, to enforce encapsulation

Domain classes

- Specific to your business domain
- E.g. BankAccount, Customer, Patient, MedicalRecord

Infrastructure classes

- Implement technical infrastructure layer
- E.g. NetworkConnection, AccountsDataAccess, IPAddress

Error classes

- Represent known types of error

# What is an Object?

An object is an instance of a class

- Created (or "instantiated") by client code
- Each object is uniquely referenced by its memory address (no need for primary keys, as in a database)

Object management

- Objects are allocated on the garbage-collected heap
- An object remains allocated until the last remaining object reference disappears
- At this point, the object is available for garbage collection
- The garbage collector will reclaim its memory sometime thereafter

# Class Diagrams

During OO analysis and design, you map the real world into candidate classes in your application.

# Defining and using a class

- General syntax for class declarations
- Creating objects
- Defining and calling methods
- Defining instance variables
- Initialization methods
- Making an object's attributes private
- Implementing method behaviour

# General syntax for class declarations

General syntax for declaring a class:

```
1  class ClassName :  
2      #  
3      # Define attributes (data and methods) here.  
4      #
```

Example:

```
1  class BankAccount :  
2      #  
3      # Define BankAccount attributes (data and methods) here.  
4      #
```

# Creating objects

To create an instance (object) of the class:

- Use the name of the class, followed by parentheses
- Pass initialization parameters if necessary (see later)
- You get back an object reference, which points to the object in memory

```
1   objectRef = ClassType(initializationParams)
```

## Example

```
1   from accounting import BankAccount  
2  
3   acc1 = BankAccount()  
4   acc2 = BankAccount()
```

# Defining and calling methods

You can define methods in a class

- i.e. functions that operate on an instance of a class

In Python, methods must receive an extra first parameter

- Conventionally named self
- Allows the method to access attributes in the target object

```
1  class BankAccount :  
2      def deposit(self, amount):  
3          print("TODO: implement deposit() code")  
4  
5      def withdraw(self, amount):  
6          print("TODO: implement withdraw() code")
```

Client code can call methods on an object

```
1  acc1 = BankAccount()  
2  acc1.deposit(200)  
3  acc1.withdraw(50)
```

# Initialization methods (1/2)

You can implement a special method named ``__init__()```

- Called automatically by Python, whenever a new object is created
- The ideal place for you to initialize the new object!
- Similar to constructors in other OO languages

Typical approach:

- Define an ``__init__()``` method, with parameters if needed
- Inside the method, set attribute values on the target object
- Perform any additional initialization tasks, if needed

Client code:

- Pass in initialization values when you create an object

# Initialization methods (2/2)

Here's an example of how to implement `\_\_init\_\_()`

```
1 class BankAccount:  
2  
3     def __init__(self, accountHolder="Anonymous"):  
4         self.accountHolder = accountHolder  
5         self.balance = 0.0  
6  
7     ...
```

This is how client code creates objects now

```
1 acc1 = BankAccount("Fred")  
2 acc2 = BankAccount("Wilma")
```

# Making an object's attributes private

One of the goals of OO is encapsulation

- Keep things as private as possible

However, attributes in Python are public by default

- Client code can access the attributes freely!

```
1 acc1 = BankAccount("Fred")
2 print("acc1 account holder is %s" % acc1.accountHolder)
```

To make an object's attributes private:

- Prefix the attribute name with two underscores, `\_\_`

```
1 class BankAccount:
2
3     def __init__(self, accountHolder="Anonymous"):
4         self.accountHolder = accountHolder
5         self.__balance = 0.0
6
7     ...
```

# Implementing method behaviour

Here's a more complete implementation of our class

```
1  class BankAccount:  
2      """Simple BankAccount class"""  
3  
4      def __init__(self, accountHolder="Anonymous"):  
5          self.accountHolder = accountHolder  
6          self.__balance = 0.0  
7  
8      def deposit(self, amount):  
9          self.__balance += amount  
10         return self.__balance  
11  
12     def withdraw(self, amount):  
13         self.__balance -= amount  
14         return self.__balance  
15  
16     def toString(self):  
17         return "{0}, {1}".format(self.accountHolder, self.__balance)
```

# Class-wide members

- Class-wide variables
- Class-wide methods
- `@classmethod` and `@staticmethod`

# Class-wide variables (1/2)

Class-wide variables belong to the class as a whole

- Allocated once, before usage of first object
- Remain allocated regardless of number of objects

To define a class-wide variable:

- Define the variable at global level in the class

```
1  class BankAccount:  
2      __nextId = 1  
3      __OVERDRAFT_LIMIT = -1000  
4      ...
```

To access the class-wide variable in methods:

- Prefix with the class name

# Class-wide variables (2/2)

Here's an example that puts it all together

```
1  class BankAccount:  
2  
3      __nextId = 1  
4      __OVERDRAFT_LIMIT = -1000  
5  
6  
7      def __init__(self, accountHolder="Anonymous"):  
8          self.accountHolder = accountHolder  
9          self.__balance = 0.0  
10         self.id = BankAccount.__nextId  
11         BankAccount.__nextId += 1  
12  
13  
14      def withdraw(self, amount):  
15          newBalance = self.__balance - amount  
16          if newBalance < BankAccount.__OVERDRAFT_LIMIT:  
17              print("Insufficient funds to withdraw %f" % amount)  
18          else:  
19              self.__balance = newBalance  
20          return self.__balance  
21  
22  ...
```

# Class-wide methods

Typical uses for class-wide methods:

- Get/set class-wide variables
- Factory methods, responsible for creating instances
- Instance management, keeping track of all instances

Example:

```
1  class BankAccount:  
2  
3      __nextId = 1  
4      __OVERDRAFT_LIMIT = -1000  
5      ...  
6  
7      def getOverdraftLimit():  
8          return BankAccount.__OVERDRAFT_LIMIT
```

Client code:

```
1  print("Overdraft limit for all accounts is %d" % BankAccount.getOverdraftLimit())
```

# @classmethod and @staticmethod

The @classmethod and @staticmethod decorators can be applied to class-wide methods

```
1  class BankAccount:  
2  
3      __OVERDRAFT_LIMIT = -1000  
4  
5      ...  
6  
7      @classmethod  
8      def getOverdraftLimit(cls):  
9          return cls.__OVERDRAFT_LIMIT  
10  
11     @staticmethod  
12     def getBanner():  
13         return "\nThis is the BankAccount Banner"
```

```
1  # Invoking via the class  
2  print(BankAccount.getBanner())  
3  print(BankAccount.getOverdraftLimit())  
4  
5  # Invoking via an instance  
6  acc1 = BankAccount("Luke")  
7  print(acc1.getBanner())  
8  print(acc1.getOverdraftLimit())
```

Any questions?

# Exceptions

- Getting started with exceptions
- Additional exception techniques

# Getting started with exceptions

- Overview
- Standard exceptions in Python
- Simple exception example
- Accessing the exception object

# Overview

Exceptions are a run-time mechanism for indicating exceptional conditions in Python

- If you detect an "exceptional" condition, you can throw an exception
- An exception is an object that contains relevant error info

Somewhere up the call stack, the exception is caught and dealt with

- If the exception is not caught, your application terminates

# Standard exceptions in Python

There are lots of things that can go wrong in a Python app

- Therefore, there are lots of different exception classes
- Each exception class represents a different kind of problem

Here are some of the standard exception classes in Python:

- KeyboardInterrupt
- OSError
- EOFError
- ValueError
- ... etc.

# Simple exception example

Here's a simple example of how to deal with exceptions in a Python app

- The try block contains code that might cause an exception
- The except block catches a particular type of exception

```
1 # Keep on looping until the user enters a number.  
2  
3 while True:  
4  
5     try:  
6         inp = input("What's your favourite number? ")  
7         num = int(inp)  
8         print("Thanks, your favourite number is %d" % num)  
9         break  
10  
11    except ValueError:  
12        print("Eek, that's not valid a number!")
```

# Accessing the exception object

In your except clause, you can specify a name for the exception object you just caught

- Allows you to use the exception object in your except block

## Example

- Catch ValueError and display error message on console

```
1 # Keep on looping until the user enters a number.
2 while True:
3     try:
4         inp = input("What's your favourite number? ")
5         num = int(inp)
6         print("Thanks, your favourite number is %d" % num)
7         break
8
9     except ValueError as err:
10        print("ValueError occurred: %s" % err)
```

## 2. Additional Exception Techniques

- Catching multiple exception types
- The "all ok" scenario
- Unconditional "wrap-up" code
- Exception hierarchies
- Defining custom exception classes
- Raising exceptions

# Catching multiple exception types (1/2)

If your try block contains complex code, then multiple different types of exception might occur

- You can define multiple except blocks, to catch each type of error
- Optionally the last except block can be a catch-all (omit the type)

Example

```
1 import sys
2
3 try:
4     fh = open('favNum.txt')
5     str = fh.readline()
6     num = int(str.strip())
7     print("The number in the file is %d" % num)
8
9 except OSError as err:
10    print("OSError occurred: %s" % err)
11
12 except ValueError as err:
13    print("ValueError occurred: %s" % err)
14
15 except:
```

# Catching multiple exception types (2/2)

If you want to perform the same processing for several types of exception:

- Group the exceptions together in a single except block
- Specify the exception types as a tuple

```
1 import sys
2
3 try:
4     fh = open('favNum.txt')
5     str = fh.readline()
6     num = int(str.strip())
7     print("The number in the file is %d" % num)
8
9 except (OSError, ValueError) as err:
10    print("Error occurred: %s" % err)
11
12 except:
13    print("Some other error occurred")
```

# The "all ok" scenario

You can add an else block at the end of try...except

- Executed only if the try block completed successfully

```
1 import sys
2
3 try:
4     fh = open('favNum.txt')
5     str = fh.readline()
6     num = int(str.strip())
7     print("The number in the file is %d" % num)
8
9 except OSError as err:
10    print("OSError occurred: %s" % err)
11 ...
12
13 else:
14    print("All completed OK!")
15    fh.close()
```

# Unconditional "wrap-up" code

You can add a finally block at the end of everything

- Always executed at the end of the try...except...else construct
- Whether an exception occurred or not

```
1 import sys
2
3 try:
4     fh = open('favNum.txt')
5     str = fh.readline()
6     num = int(str.strip())
7     print("The number in the file is %d" % num)
8
9 except OSError as err:
10    print("OSError occurred: %s" % err)
11 ...
12
13 else:
14    print("All completed OK!")
15    fh.close()
16
17 finally:
18    print("That's all folks. This message will always appear!")
```

# Exception hierarchies (1/2)

Python organizes exceptions into an inheritance hierarchy

- Represents specializations of general error conditions

Example

- There are several subclasses of OSError
- BaseException
  - Exception
    - OSError
      - FileNotFoundError
      - FileExistsError
      - PermissionError
      - ChildProcessError

# Exception hierarchies (2/2)

When you define an except block...

- It will catch that exception type, plus any subclasses

Example:

- "Special" processing for FileNotFoundError exceptions
- "Generic" processing for any other kind of OSError exceptions

```
1 import sys
2
3 try:
4     fh = open('favNum.txt')
5     str = fh.readline()
6     num = int(str.strip())
7     print("The number in the file is %d" % num)
8
9 except FileNotFoundError as err:
10    print("File not found: %s" % err)
11
12 except OSError as err:
13    print("More general OSError occurred: %s" % err)
```

# Defining custom exception classes

You can define custom exception classes

- To represent important types of error in your application

How to do it:

- Define a class that inherits from `Exception` (or a subclass)
- Implement `__init__` and `__str__` methods

Example:

```
1  class MyError(Exception):
2
3      def __init__(self, value):
4          self.value = value
5
6      def __str__(self):
7          return repr(self.value)
```

# Raising exceptions

To raise (i.e. trigger) an exception:

- Use the `raise` keyword
- Specify the type of exception you want to raise
- Pass in any constructor arguments as appropriate

Example:

```
1  try:  
2      raise MyError("EEK ERROR ERROR ERROR")  
3  
4  except MyError as err:  
5      print("It appears my exception occurred, the value is %s" % err.value)
```

Any questions?

# Bonus

- Sharing modules/packages with friends

# Sharing modules/packages with friends

- virtualenv
- requirements.txt
- Freezing
- Installing

# virtualenv

These can be managed within PyCharm or manually yourself.

It's probably best to allow PyCharm to do the heavy lifting here.

# requirements.txt

You can see the list of currently installed 3rd party packages with

```
1 pip freeze
```

To share this with someone else, conventionally we save this to a text file:

```
1 pip freeze > requirements.txt
```

and store this text file in source control.

# Installing from requirements.txt

To install from a requirements file, make sure you have the right virtualenv setup.

If necessary, create a new one with PyCharm or the command line tool.

Install the dependencies with:

```
1 pip install -r requirements.txt
```

Run the code as normal.

Note: This will synchronise the requirements but will still need you and your colleague to have a compatible version of the Python interpreter running.

# More Object Oriented Programming (OOP)

- A closer look at attributes
- Implementing magic methods
- Inheritance
- Help documentation
- Copying object state
- Reading/writing objects to a file

# A closer look at attributes

- Determining an object's attributes
- Adding and removing object attributes
- Built-in class attributes

# Determining an object's attributes

Python provides several global functions that allow you to manage attributes on an object

```
1 from accounting import BankAccount
2
3 acc1 = BankAccount("Fred")
4
5 setattr(acc1, "bonus", 2000)
6
7 if hasattr(acc1, "bonus"):
8     print("acc1.bonus is %d" % acc1.bonus)
9
10 delattr(acc1, "bonus")
```

# Adding and removing object attributes

You can also add and remove attributes on an object directly, as follows:

```
1 from accounting import BankAccount
2
3 acc1 = BankAccount("Fred")
4
5 # Add an attribute to an object.
6 acc1.flag = "Whao watch this guy"
7 print("acc1.flag is %s" % acc1.flag)
8
9 # Remove an attribute from an object.
10 del acc1.flag
```

# Built-in class attributes

Every class provides metadata via the following built-in attributes

- You can also get metadata about an object too

```
1  from accounting import BankAccount
2
3  print("BankAccount.__doc__:",     BankAccount.__doc__)
4  print("BankAccount.__name__:",    BankAccount.__name__)
5  print("BankAccount.__module__:",  BankAccount.__module__)
6  print("BankAccount.__bases__:",   BankAccount.__bases__)
7  print("BankAccount.__dict__:",    BankAccount.__dict__)
8
9  acc1 = BankAccount("Ola")
10 print("acc1.__dict__:", acc1.__dict__)
```

# Implementing magic methods

- Overview
- Implementing constructors and destructors
- Implementing stringify methods
- Implementing operator methods

# Overview

There are various "special" methods you can implement in your Python classes

- These methods allow your class objects to take advantage of standard Python idioms

It's good practice to implement these methods where relevant

- Python programmers will recognise these methods immediately
- Makes your classes easier to maintain

# Implementing constructors and destructors

## Constructor

- `\_\_init\_\_(self, otherArgs)`

## Destructor

- `\_\_del\_\_(self)`

## Example

```
1  class Person:  
2  
3      def __init__(self, name, age):  
4          self.name = name  
5          self.age = age  
6          print("In __init__() for %s and %d" % (self.name, self.age))  
7  
8      def __del__(self):  
9          print("In __del__() for %s and %d" % (self.name, self.age))  
10  
11     p1 = Person("Bill", 23)  
12     p2 = Person("Ben", 25)
```

# Implementing stringify methods

Return a machine-readable representation of an object

- `\_\_repr\_\_(self)`

Return a human-readable representation of an object

- `\_\_str\_\_(self)`

Example

```
1  class Person:  
2  
3      def __repr__(self):  
4          return "{0} instance, name: {1}, age: {2}".format( \  
5                      self.__class__.__name__, \  
6                      self.name, self.age)  
7      def __str__(self):  
8          return "{0} is {1}.".format(self.name, self.age)  
9      ...  
10  
11  print(repr(p1))  
12  print(str(p2))
```

# Implementing operator methods

There are a large number of method that represent standard operators, including:

- `\_\_eq\_\_(self, other)`
- `\_\_ne\_\_(self, other)` Etc...

## Example

```
1  class Person:  
2  
3      def __eq__(self, other):  
4          return self.age == other.age  
5  
6      def __ne__(self, other):  
7          return self.age != other.age  
8  
9      ...  
10     ...  
11  
12  
13     print("p1 == p2 gives %s" % (p1 == p2))  
14     print("p1 != p2 gives %s" % (p1 != p2))
```

# Inheritance

- Overview of inheritance
- Superclasses and subclasses
- Sample hierarchy
- Defining a subclass
- Adding new members
- Defining constructors
- Overriding methods
- Multiple inheritance

# Overview of inheritance

Inheritance is a very important part of object-oriented development

- Allows you to define a new class based on an existing class
- You just specify how the new class differs from the existing class

Terminology:

- For the "existing class": Base class, superclass, parent class
- For the "new class": Derived class, subclass, child class

Potential benefits of inheritance:

- Improved OO model
- Faster development
- Smaller code base

# Superclasses and subclasses

The subclass inherits everything from the superclass (except constructors)

- You can define additional variables and methods
- You can override existing methods from the superclass
- You typically have to define constructors too
- Note: You can't cherry pick or "blank off" superclass members

# Sample hierarchy

We'll see how to implement the following simple hierarchy:

```
BankAccount <- SavingsAccount
```

Note:

- BankAccount defines common state and behaviour that is relevant for all kinds of account
- SavingsAccount "is a kind of" BankAccount that earns interest

We might define additional subclasses in the future...

- E.g. CurrentAccount, a kind of BankAccount that has cheques

# Defining a subclass

To define a subclass, use the following syntax

- Note that a Python class can inherit from multiple superclasses
- We'll discuss multiple inheritance later in this chapter

```
1  class Subclass(Superclass1, Superclass2, ...):  
2  
3      # Additional attributes and methods ...  
4  
5      # Constructor(s) ...  
6  
7      # Overrides for superclass methods, if necessary ...
```

Example:

```
1  class SavingsAccount(BankAccount):  
2      ...  
3      ...  
4      ...
```

# Adding new members

The subclass inherits everything from the superclass

- (Except for constructors)
- The subclass can define additional members if it needs to ...

Example:

```
1  class SavingsAccount(BankAccount):  
2  
3      __DEFAULT_INTEREST_RATE = 1.5  
4  
5  
6      def earnInterest(self):  
7          self.balance *= (1 + self.interestRate)  
8          return self.balance  
9  
10     ...
```

# Defining constructors

A subclass doesn't inherit the constructor from superclass

- So, define a constructor in the subclass, to initialize subclass state

The subclass constructor should invoke the superclass constructor, to initialize superclass data

- Call `super().\_\_init\_\_(params)`

Example:

```
1  class SavingsAccount(BankAccount):  
2  
3      def __init__(self, accountHolder="Anonymous", interestRate=None):  
4  
5          super().__init__(accountHolder)  
6  
7          if interestRate is None:  
8              self.interestRate = SavingsAccount.__DEFAULT_INTEREST_RATE  
9          else:  
10              self.interestRate = interestRate  
11  
12      ...
```

# Overriding methods

The subclass can override superclass instance methods

- To provide a different (or supplementary) implementation
- No obligation ☺

An override can call the original superclass method, to leverage existing functionality

- Call super().methodName(params)

Example:

```
1  class SavingsAccount(BankAccount):  
2  
3      def withdraw(self, amount):  
4          if amount > self.balance:  
5              print("You can't go overdrawn in a savings account!")  
6          else:  
7              super().withdraw(amount)  
8          return self.balance  
9  
...
```

# Multiple inheritance (1/2)

Python supports multiple inheritance

```
1 class Logger:  
2     def log(self, msg):  
3         print(msg)  
4  
5 class Beeper:  
6     def beep(self, duration):  
7         winsound.Beep(2500, duration)
```

```
1 class Alerter(Logger, Beeper):  
2     def doShortAlert(self, msg):  
3         super().log(msg)  
4         super().beep(250)  
5  
6     def doMediumAlert(self, msg):  
7         super().log(msg)  
8         super().beep(1000)  
9  
10    def doLongAlert(self, msg):  
11        super().log(msg)  
12        super().beep(2500)
```

# Multiple inheritance (2/2)

Client code can access public members in the subclass or in any superclass

```
1 alerter = Alerter()
2
3 alerter.log("Wakey wakey!")
4 for i in range(30):
5     alerter.beep(50)
6
7 msg = input("Enter an alert message: ")
8 alerter.doShortAlert(msg)
9
10 msg = input("Enter another alert message: ")
11 alerter.doMediumAlert(msg)
12
13 msg = input("And another: ")
14 alerter.doLongAlert(msg)
```

# Help documentation

You can provide "help" documentation at the start of the class and the start of each method

- Define a help string """like this"""
- For an example, see accounting2.py

You can then get help for the class or methods via the help() function in the Python shell

# Copying object state

When you assign one object reference to another:

- It just copies the object reference
- So both references refer to the same actual object

If you want to create a copy of an object:

- Call the `copy()` function, defined in the `copy` module

Example: See `demoCopying.py`

# Reading/writing objects to a file

A common requirement is to read/write objects to a file

There are various ways to do this in Python:

- As JSON
- As XML
- As CSV

You can also write your own custom code See `accounting3.py`, `clientcodeReadWriteObjects.py`

Any questions?

# Functional Programming

- Functional programming in Python
- Higher order functions
- Additional techniques

# 1. Functional Programming in Python

- Overview of functional programming (FP)
- Function evaluation
- Pure functions
- Anonymous functions a.k.a. lambdas
- Lambda example
- Lambdas and parameters

# Overview of functional programming (FP)

FP is a style of programming characterised by...

- Treating computation as the evaluation of functions
- Use of higher-order functions and/or recursion
- Immutable (read-only) state
- Lazy evaluation

Why use FP?

- Very amenable to multi-threading
- Share complex algorithms across multiple threads, to maximise concurrency and increase performance

Any disadvantages?

- Quite a steep learning curve
- Not suitable for every problem

# Function evaluation

Functions depend only on their inputs, and not on other program state

For example, consider the following function

```
1 def cube(x):  
2     return x * x * x
```

It always gives the same answer for the same input (so it has predictable behaviour - you can reason about its operation)

It has no local state, side-effects, or changes to any other program state (so it can be safely executed by multiple threads)

# Pure functions

A "pure function" is one that has no side effects. This has several useful consequences:

- If the result of a pure expression is not used, it can be removed without affecting anything else
- If a pure function is called with the same arguments, you will get the same result (so evaluations can be cached)
- If there is no data dependency between two pure functions, they can be evaluated in any order, or performed in parallel

# Anonymous functions a.k.a. lambdas

A lambda expression is a 1-line inline expression

- Like an anonymous function

To define a lambda expression:

- Use the `lambda` keyword
- Followed by the argument list
- Followed by a colon
- Followed by a 1-line inline expression

```
1 my_lambda = lambda arg1, arg2, ... argn : inline_expression
```

To invoke a lambda expression:

- Same syntax as a regular function call

```
1 my_lambda(argvalue1, argvalue2, ..., argvaluen)
```

# Lambda example

A lambda that takes a single parameter and returns the square of that value

```
1 mylambda = lambda x: x * x
2
3 result = mylambda(10)
4 print(result)
```

# Lambdas and parameters

Lambdas can take multiple parameters

- List all the parameters after the lambda keyword

```
1 mylambda = lambda x, y: print(f"You passed {x}, {y}")
2 mylambda(10, 20)
```

Lambdas can take no parameters

- Just follow the lambda keyword with a : immediately

```
1 mylambda = lambda: print("Hello!")
2
3 mylambda()
```

# 2. Higher Order Functions

- Overview of higher-order functions
- Passing a lambda to a function
- Returning a lambda from a function
- Closures

# Overview of higher-order functions

Higher-order functions can use other functions as arguments and return values

- You can pass a function as a parameter into another function
- You can return a function from a function

We'll explore both these techniques in the following slides

- We'll use lambdas to represent the function parameters/returns

# Passing a lambda to a function

You can pass a lambda as a parameter into a function

- Allows you to write very generic functions

Example

- The apply() function applies the lambda that you pass in

```
1 def apply(arg1, arg2, op) :  
2     return op(arg1, arg2)  
3  
4 result1 = apply(10, 20, lambda x, y: x + y)  
5 print(result1)  
6  
7 result2 = apply(10, 20, lambda x, y: x / y)  
8 print(result2)
```

# Returning a lambda from a function

You can return a lambda from a function

Consider this simple concat() function

- Concatenates its two parameters in the order specified

```
1 def concat(str1, str2):  
2     return str1 + str2
```

Now consider the flip() function

- Takes a binary operation
- Returns a lambda that performs the operation with args flipped

```
1 def flip(binaryOp) :  
2     return lambda x, y: binaryOp(y, x)  
3  
4 # Usage.  
5 flipConcat = flip(concat)  
6 result2 = flipConcat("Hello", "World")  
7 print(result2)
```

# Closures (1/2)

A closure is a function whose behaviour depends on variables declared outside the scope in which it is then used

- This is often used when returning functions/lambdas
- The returned function/lambda remembers the original state in the enclosing function

```
1 def banner(start, end) :  
2     return lambda msg: print(f"{start} {message} {end}")  
3  
4 bannerMsg = banner(" [ --- ", " --- ]")  
5  
6 bannerMsg("Hello")  
7 bannerMsg("World")
```

# Closures (2/2)

Here, fib returns a function that calculates Fibonacci numbers, returns the next one each time called

```
1 def fib():
2     tup = (1, -1)
3     def retfunc():
4         nonlocal tup
5         tup = (tup[0] + tup[1], tup[0])
6         return tup[0]
7
8     return retfunc
```

Note 1: nonlocal keyword lets you access a variable in external scope

Note 2: tup is a tuple, and you access its members using [0] and [1]

# 3. Additional Techniques

- Recursion
- Tail recursion
- Reduction
- Partial functions

# Recursion

Recursion is commonly used instead of looping

- It avoids the mutable state associated with loop counters

```
1 def factorial(n):
2     if n == 0:
3         return 1
4     else:
5         return n * factorial(n - 1)
```

```
1 result = factorial(4)
2 print("4 factorial is %d\n" % result)
```

# Tail Recursion

Tail recursion is where the very last thing you do in a function is call yourself

- The function calls can theoretically be executed in a simple loop

Here's a tail-recursive implementation of factorial

```
1 def tailRecursiveFactorial(accumulator, n):  
2     if n == 0:  
3         return accumulator  
4     else:  
5         return tailRecursiveFactorial(n * accumulator, n - 1)  
6  
7 result = tailRecursiveFactorial(1, 4)  
8 print("4 factorial is %d\n" % result)
```

# Reduction

The `functools` module has several useful utility functions for functional programming

- E.g. `reduce()`, which reduces the elements in a collection to a single result

```
1 from functools import reduce
2
3 mylambda = lambda x,y: x+y
4
5 result = reduce(mylambda, [3,12,19,1,2,7])
6 print(result)
```

# Partial Functions

The `functools` module also allows you to create partial functions, i.e. functions with one or more args already filled in

- Via `partial()`

```
1  from functools import partial
2
3  multiply = lambda x,y: x * y
4
5  times2 = partial(multiply, 2)
6  times5 = partial(multiply, 5)
7  times8 = partial(multiply, 8)
8
9  print("10 times 2 is %d" % times2(10))
10 print("10 times 5 is %d" % times5(10))
```

Note: If you're interested to learn how this works, see our own version of `partial()` here:

- `PartialFunctionsHowTheyWork.py`

# Any Questions?

# Decorators

1. Getting started with decorators
2. Additional decorator techniques
3. Parameterized decorators

# 1. Getting started with decorators

- Overview
- Defining a decorator function
- Applying a decorator function manually
- Applying a decorator function properly

# Overview

Python allows you to decorate functions and classes using the @decorator syntax

The decorator enhances the function/class with extra capabilities

```
1  @someDecorator  
2  def someFunction(...) :  
3      ...
```

```
1  @someDecorator  
2  class SomeClass :  
3      ...
```

This section shows how to define decorators

- We'll see how to define decorator functions
- We'll also describe how Python applies decorator functions

# Defining a decorator function (1/2)

Define a function that takes a function pointer as an argument

- The pointer indicates the target function you want to decorate

Inside the decorator function, implement a nested function

- The nested function should call the target function
- And should also perform the desired decoration behaviour

At the end of the decorator function, return a pointer to the nested function

```
1 def simpleDecorator(func) :  
2  
3     # Define an inner function, which wraps (decorates)  
4     def innerFunc() :  
5         print("Start of simpleDecorator()")  
6         func()  
7         print("End of simpleDecorator()")  
8  
9     # Return the inner function.  
10    return innerFunc
```

# Applying a decorator function manually

In order to understand how decorators work, let's first of all see how to apply a decorator function manually:

Line 6 calls the decorator function manually, passing a pointer to the target function as an argument

- This statement returns a pointer to the inner function

Line 7 calls the inner function

- This invokes the target function, with the desired decoration

```
1 # Some function that we want to decorate.  
2 def myfunc1() :  
3     print("Hi from myfunc1()")  
4  
5 # Client code.  
6 pointerToInnerFunc = simpleDecorator(myfunc1)  
7 pointerToInnerFunc()
```

# Applying a decorator function properly

The previous slide showed how to call a decorator function manually, to wrap a target function

Now let's see how to apply a decorator function properly, i.e. using the @decorator syntax

```
1 # Some function, which we now decorate explicitly.  
2 @simpleDecorator  
3 def myfunc1() :  
4     print("Hi from myfunc1()")  
5  
6 # Client code.  
7 myfunc1()
```

Note the client code just calls myfunc1() directly

- Python intervenes, thanks to the @simpleDecorator decorator, and converts the code to the equivalent of the previous slide

# Additional decorator techniques

- Decorating a function that takes arguments
- Decorating a function that returns a result

# Decorating a function that takes arguments

Consider the following function, which takes arguments Note that we've decorated the function

```
1 @parameterAwareDecorator
2 def myfunc1(firstName, lastName, nationality) :
3     print(f"Hi {firstName} {lastName}, your nationality is {nationality}")
```

This is how to define the decorator function

- The inner function receives variadic args and passes to target func

```
1 def parameterAwareDecorator(func) :
2
3     def innerFunc(*args, **kwargs) :
4         print("Start of parameterAwareDecorator()")
5         func(*args, **kwargs)
6         print("End of parameterAwareDecorator()")
7
8     return innerFunc
```

Client code:

```
1 myfunc1("Olaf", "Hansmann", "Norfolk")
```

# Decorating a function that returns a result

Consider the following function, which returns a result

```
1 @returnAwareDecorator
2 def myfunc1(firstName, lastName, nationality) :
3     return f"Hi {firstName} {lastName}, your nationality is {nationality}"
```

This is how to define the decorator function

- The inner function returns the result of the target function

```
1 def returnAwareDecorator(func) :
2
3     def innerFunc(*args, **kwargs) :
4         print("Start of returnAwareDecorator()")
5         returnValueFromFunc = func(*args, **kwargs)
6         print("End of returnAwareDecorator()")
7         return returnValueFromFunc
8
9     return innerFunc
```

Client code:

# Parameterized decorators

- Overview
- Defining a parameterized decorator
- Applying a parameterized decorator manually
- Applying a parameterized decorator properly

# Overview

Decorators can take parameters, to make them flexible

E.g. imagine a flexible decorator that displays custom pre/post messages around a target function call

- You might apply the decorator as follows
- The decorator takes parameters specifying the pre/post messages

```
1 @parameterizedDecorator("HELLO", "GOODBYE")
2 def myfunc1(firstName, lastName, nationality) :
3     return "Hi %s %s, your nationality is %s" % (firstName, lastName, nationality)
```

# Defining a Parameterized Decorator

Here's how to define a parameterized decorator

```
1 def parameterizedDecorator(prefix, suffix) :
2
3     # Define inner function, which just wraps a function
4     def innerFunc1(func) :
5
6         # Define inner-inner function, which decorates
7         def innerFunc2(*args, **kwargs) :
8             print(prefix)
9             returnValueFromFunc = func(*args, **kwargs)
10            print(suffix)
11            return returnValueFromFunc
12
13        # Return innerFunc2, i.e. the inner-inner function.
14        return innerFunc2
15
16    # Return innerFunc1, i.e. the inner function.
17    return innerFunc1
```

We have a layering of functions, to handle all the args:

- Arguments to the decorator itself
- The target function to be invoked
- Arguments to pass in to the target function

# Applying a Parameterized Decorator Manually

In order to understand how parameterized decorators work, let's first see how to apply the decorator manually:

```
1 # Some function, which we don't decorate explicitly here
2 def myfunc1(firstName, lastName, nationality):
3     return "Hi %s %s, your nationality is %s" % (first
4
5 # Client code
6 pointerToInnerFunc1 = parameterizedDecorator("HELLO",
7 pointerToInnerFunc2 = pointerToInnerFunc1(myfunc1)
8 res = pointerToInnerFunc2("Per", "Nordmann", "Norsk")
```

Line 6 calls the decorator manually, passing args into it

- This statement returns a pointer to innerFunc1

Line 7 calls innerFunc1, passing target function into it

- This just return a pointer to innerFunc2

Line 8 calls innerFunc2, passing args for the target func

- This invokes the target func, with the desired decoration

# Applying a Parameterized Decorator Properly

The previous slide showed how to call a parameterized decorator function manually, to wrap a target function

Now let's see how to apply a parameterized decorator function properly, i.e. using the @decorator syntax

```
1 # Some function, which we now decorate explicitly.  
2 @parameterizedDecorator("HELLO", "GOODBYE")  
3 def myfunc1(firstName, lastName, nationality) :  
4     return "Hi %s %s, your nationality is %s" % (firstName, lastName, nationality)  
5  
6 # Client code.  
7 res1 = myfunc1("Kari", "Nordmann", "Norsk")
```

Note the client code just calls myfunc1() directly

- Python intervenes, thanks to the @parameterizedDecorator, and cascades through the necessary sequence of function calls and argument-passing

Any questions?

# Asynchronous Processing in Python

1. Getting started with asynchrony in Python
2. Managing coroutines via tasks
3. Additional task techniques

# Getting started with asynchrony in Python

- Overview
- Asynchrony in Python
- Coroutines
- The asyncio module
- Simple example of asynchrony

# Overview

What is asynchrony?

- The ability to perform multiple tasks concurrently

Scenarios where asynchrony is important:

- Processing a large dataset in parallel
- Handling multiple network connections simultaneously
- Performing algorithmic processing in the background
- Etc.

# Asynchrony in Python

You can schedule concurrent tasks on a single thread

- The event loop manages task execution on a thread

The event loop can optimize I/O

- If a function is waiting on I/O
- The event loop pauses the function and runs another one instead
- When the first function completes I/O, it is resumed

The event loop can also optimize CPU-intensive functions

- The functions must explicitly "yield", so as not to hog the thread

Note: Python also supports genuine multithreading

- See `multipleThreadsAndFutures.py`

# Coroutines

A coroutine is a special kind of generator function

- It can cede control during its processing (e.g. for I/O)
- The event loop then tries to give another coroutine some time
- The event loop can resume the original coroutine when it's ready

The preferred way to define a coroutine in modern Python is to prefix a function with the `async` keyword

```
1  async def someFunc(someArgs) :  
2      # Some long-running code that might yield control  
3      #   e.g. code that does slow I/O  
4      #   e.g. code that CPU-intensive processing
```

# The asyncio module

The asyncio module provides various methods that allow you to schedule and manage asynchrony

- Some of the common methods are listed here

`asyncio.sleep(seconds)`

- Sleep for a specified delay (in seconds)

`asyncio.run(aCoroutine)`

- Creates a new event loop, and runs the coroutine

`asyncio.create_task()`

- Schedule a coroutine to be executed "soon" on the event loop

# Simple example of asynchrony

```
1 import asyncio
2 from time import strftime, localtime
3
4 async def displayAfter(msg, delay):
5     await asyncio.sleep(delay)
6     now = strftime("%H:%M:%S", localtime())
7     print("%s %s" % (now, msg))
8
9 def main():
10    print("*****Start of main*****")
11    asyncio.run(displayAfter("Hei", 3))
12    asyncio.run(displayAfter("Bye", 5))
13    print("*****End of main*****")
14
15 if __name__ == "__main__":
16     main()
```

- `asyncio.sleep()` is a coroutine
- The `await` keyword yields control back to the event loop, which tries to schedule other coroutines in the meantime
- You can only use the `await` keyword in coroutines, i.e. functions marked as `async`
- You can't just 'invoke' coroutines, you must schedule via `asyncio`

# Managing coroutines via tasks

- Overview
- Simple example of creating a task
- Creating and awaiting multiple tasks
- Awaiting multiple tasks to complete

# Overview

The `asyncio.create_task()` function creates a task

- The task is scheduled for execution "soon" on the event loop
- The task is represented by a Task object

The Task class has methods that allow you to manage the running of the task, such as:

- `done()` - has the task completed yet?
- `cancel()` - stop the task now
- `result()` - get the result of the task (it must have finished!)

# Simple example of creating a task

```
1  from time import strftime, localtime
2  import asyncio
3
4  def doDisplay(msg):
5      now = strftime("%H:%M:%S", localtime())
6      print("%s %s" % (now, msg))
7
8  async def displayAfter(msg, delay) :
9      doDisplay("START: " + msg)
10     await asyncio.sleep(delay)
11     doDisplay("END: " + msg)
12
13 async def main():
14     print("*****Start of main*****")
15     task = asyncio.create_task(displayAfter("Hello", 10))
16
17     for i in range(0,5) :
18         print("Doing something useful ... ")
19         await asyncio.sleep(1)
20
21     print("Finished doing useful work, now I'll wait for task to finish")
22     await task
23     print("*****End of main*****")
24
25 if __name__ == "__main__":
26     main()
```

# Creating and awaiting multiple tasks

You can create multiple tasks

- All the tasks run concurrently
- You can await for each task to complete individually

```
1  async def main():
2      doDisplay("*****Start of main*****")
3      task1 = asyncio.create_task(displayAfter("Bonjour"))
4      task2 = asyncio.create_task(displayAfter("Bore da"))
5      task3 = asyncio.create_task(displayAfter("Hei hei"))
6
7      for i in range(0,5) :
8          doDisplay("Doing something useful ... ")
9          await asyncio.sleep(1)
10
11     doDisplay("Waiting for task1 to finish")
12     await task1
13
14     doDisplay("Waiting for task2 to finish")
15     await task2
16
17     doDisplay("Waiting for task3 to finish")
18     await task3
19
20     doDisplay("*****End of main*****")
```

# Awaiting multiple tasks to complete

The previous example awaited individual tasks to complete

- If you prefer, you can await multiple tasks to complete
- Use `asyncio.gather()`, which suspends until all tasks are done

```
1  async def main():
2      doDisplay("*****Start of main*****")
3      task1 = asyncio.create_task(displayAfter("Bonjour", 10))
4      task2 = asyncio.create_task(displayAfter("Bore da", 15))
5      task3 = asyncio.create_task(displayAfter("Hei hei", 20))
6
7      for i in range(0,5) :
8          doDisplay("Doing something useful ... ")
9          await asyncio.sleep(1)
10
11     doDisplay("Waiting multiple tasks to complete")
12     await asyncio.gather(task1, task2, task3)
13
14     doDisplay("*****End of main*****")
```

# Additional task techniques

- Awaiting the result of a task
- Polling a task to see if it's done
- Cancelling a task

# Awaiting the result of a task (1/2)

A coroutine can return a value

- The calling code would like to retrieve the value when complete

Here's one way for the calling code to do this:

- Create a task, to schedule the coroutine for execution
- Await completion of the task
- The await expression gives the result of the completed coroutine

```
1  async def createStringAfter(msg, delay) :
2      await asyncio.sleep(delay)
3      now = strftime("%H:%M:%S", localtime())
4      return "{0} {1}".format(now, msg)
5
6  async def main():
7      print("*****Start of main*****")
8      task = asyncio.create_task(createStringAfter("Bonjour", 10))
9      result = await task
10     print(result)
11     print("*****End of main*****")
```

# Awaiting the result of a task (2/2)

The previous slide created a task, and then awaited its completion separately:

```
1  async def main():
2      print("*****Start of main*****")
3      task = asyncio.create_task(createStringAfter("Bonjour", 10))
4      result = await task
5      print(result)
6      print("*****End of main*****")
```

If it's more convenient, you can combine these two statements into a single statement

```
1  async def main():
2      print("*****Start of main*****")
3      result = await asyncio.create_task(createStringAfter("Bonjour", 10))
4      print(result)
5      print("*****End of main*****")
```

# Polling a task to see if it's done

Sometimes you might want to poll a task to see if it's done

- Call done() on the task, to see if it's finished
- If it hasn't finished, do something else for a bit, then check again
- When it really has finished, call result() on the task

```
1  async def main():
2      doDisplay("*****Start of main*****")
3      task = asyncio.create_task(createStringAfter("Bonjour", 10))
4
5      while True:
6          if task.done():
7              result = task.result()
8              doDisplay(result)
9              break
10         else:
11             doDisplay("Doing something useful ... ")
12             await asyncio.sleep(1)
13
14     doDisplay("*****End of main*****")
```

# Cancelling a task

Sometimes you might want to cancel a task mid-flight

- Call `cancel()` on the task

```
1  async def main():
2      doDisplay("*****Start of main*****")
3      task = asyncio.create_task(createStringAfter("Bonjour", 10))
4
5      while True:
6          if task.done():
7              result = task.result()
8              doDisplay(result)
9              break
10         else:
11             cancel = input("Task not complete yet. Do you want to cancel it? ")
12             if cancel == "y":
13                 doDisplay("OK I'll cancel the task and we'll all just move on in life.")
14                 task.cancel()
15                 break
16             else:
17                 doDisplay("OK I'll wait another second and do something useful ... ")
18                 await asyncio.sleep(1)
19
20     doDisplay("*****End of main*****")
```

Any questions?

# Getting Started with NumPy

- Introduction to Python data science
- NumPy arrays
- Manipulating array elements
- Manipulating array shape

# Introduction to Python data science

- Overview
- Python libraries for data science
- Getting the data science libraries

# Overview

Python is a popular choice for data science and machine learning

Attractive characteristics of Python:

- Dynamic language, so it's good for rapid exploratory coding
- Relatively simple syntax, so it's easier to become proficient
- Popular in schools and universities, so the skills are out there!

# Python libraries for data science

NumPy is a numeric processing API for Python

- Fast mathematical computation of numeric arrays and matrices

Pandas provides additional features based on NumPy

- Additional support for indexing, reading/writing CSV/Excel, etc.

Matplotlib is a graphical plotting API for Python

- Similar to Matlab, allows you to plot graphs, charts, etc.

Scikit-Learn is a machine learning library for Python

- Implements many supervised/unsupervised learning algorithms

# Getting the data science libraries

If you're using Anaconda, these are already downloaded for you.

If you're using a standalone Python distribution, you'll need to install the libraries you need.

```
1 pip install numpy  
2  
3 pip install openpyxl  
4 pip install xlrd  
5 pip install matplotlib  
6  
7 pip install pandas
```

# NumPy arrays

- Getting Started with NumPy arrays
- Techniques for creating NumPy arrays
- Reading CSV data
- Visualizing data

# Getting Started with NumPy arrays (1/2)

NumPy holds data in N-dimensional arrays

- An array is an instance of the `numpy.ndarray` class
- <https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html>

All the data in a NumPy array is the same type

- This allows NumPy to store and process the data efficiently
- This is a very important point!

Why are NumPy arrays more efficient than Python lists?

- Python is dynamically typed, so every object contains metadata that identifies the type at run time
- In a Python list, every item contains this metadata - eek!
- In a NumPy array, only the array itself contains the metadata

# Getting Started with NumPy arrays (2/2)

## Example

- Creates a NumPy array from a Python list
- Gets the shape of the array, via the `shape` property
- Gets the data type of array elements, via the `dtype` property

```
1 import numpy as np
2
3 # Create a 1D NumPy array from a Python list.
4 a = np.array([1, 2, 3])
5 print('Data values in a\n', a)
6 print('Shape of a:', a.shape)
7 print('Data type in a:', a.dtype)
8
9 # Create a 2D Numpy array from a Python list of lists.
10 b = np.array([[1, 2, 3], [4, 5, 6]])
11 print('\nData values in b\n', b)
12 print('Shape of b:', b.shape)
13 print('Data type in b:', b.dtype)
```

For a full list of NumPy standard types, see:

- <https://numpy.org/devdocs/user/basics.types.html>

# Techniques for creating NumPy arrays (1/2)

There are lots of ways to create a NumPy array

```
1 import numpy as np
2
3 a = np.array([1, 2, 3.14]) # Create array with mixed types - NumPy converts elements "upwards".
4 b = np.array([1, 2, 3], dtype='float64') # Create array with a specified type.
5 c = np.arange(0, 20, 2) # Create array from a numeric range.
6 d = np.linspace(0.0, 1.0, 11) # Create array of elements, linear spaced.
7 e = np.zeros(5) # Create array of zeros.
8 f = np.ones(5) # Create array of ones.
9 g = np.full(5, 1.23) # Create array of elements, with specified value.
10 h = np.empty(5) # Create array of elements, no specified value.
```

# Techniques for creating NumPy arrays (2/2)

You can also create random arrays, which can be handy

```
1 # Import the NumPy module.  
2 import numpy as np  
3  
4 # Create random values in range [0.0, 1.0].  
5 a = np.random.random(10)  
6  
7 # Create normally-distributed random values.  
8 b = np.random.normal(5, 2, 10)  
9  
10 # Create random integers in range [0, 101].  
11 c = np.random.randint(0, 101, 10)
```

# Reading CSV data

A common requirement is to read data from a CSV file

- The easiest way to do this is via the Pandas `read_csv()` function

Pandas reads values into a multi-column DataFrame

- You can then extract a column into a NumPy array

```
1 import numpy as np
2 import pandas as pd
3
4 # Read a csv file, get a Pandas DataFrame back.
5 dataframe = pd.read_csv('WorldCupWinners.csv')
6
7 # Get the 'Teams' column.
8 teams = np.array(dataframe['Team'])
9 print(teams)
```

We'll not have time to dive into Pandas on this course but it's a topic worth exploring!

# Visualizing data (1/2)

Visualization is an important aid to help you understand the shape and meaning of data

You can use the MatPlotLib library to visualize data in lots of different ways

- Line graphs
- Scatter graphs
- Bar-charts
- Pie-charts
- Histograms
- Etc.

# Visualizing data (2/2)

Here's a simple example of how to visualize data using Matplotlib - we'll see more plotting features later

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4
5 data = np.array([1, 19, 76, 45, 34, 42, 30, 5, 77, 54, 89])
6
7 plt.xlabel('Element in array')
8 plt.ylabel('Value')
9 plt.plot(data)
10 plt.show()
```

# Manipulating array elements

- Indexing into an array
- Slicing an array
- Accessing a specific column or row
- Aside: Views vs. copies

# Indexing into an array

Indexing into a NumPy array is quite intuitive

- [i] Access element from start, first element is at [0]
- [-i] Access element from end, last element is at [-1]
- [r,c] Access element in 2-D array (etc. for higher dimensions)

```
1 import numpy as np
2
3 # Create a 1-D array, index into it, and modify element
4 a = np.array([0, 10, 20, 30, 40, 50, 60, 70])
5 print(a)
6 print(a[1])          # 10
7 print(a[-1])         # 70
8 a[1] = 111
9 print(a)
10
11 # Create a 2-D array, index into it, and modify element
12 b = np.array([[0, 10, 20, 40], [50, 60, 70, 80]])
13 print(b)
14 print(b[0, 1])      # 10
15 print(b[0, -1])    # 40
16 print(b[-1, 1])    # 60
17 print(b[-1, -1])   # 80
18 b[0, 1] = 111
19 print(b)
```

# Slicing an array

You can slice into an array using a [start:stop:step] index

- start Default start is 0
- stop Default stop is the size of the dimension
- step Default step is 1

```
1 import numpy as np
2
3 # Create a 1-D array, and get various slices.
4 a = np.array([0, 10, 20, 30, 40, 50, 60, 70, 80])
5 print(a[3:])          # [30 40 50 60 70 80]
6 print(a[:3])          # [ 0 10 20]
7 print(a[3:7])         # [30 40 50 60]
8 print(a[3:7:2])       # [30 50]
9 print(a[3::2])        # [30 50 70]
10 print(a[ ::2])       # [ 0 20 40 60 80]
11
12 # Create a 2-D array, and get various slices in each c
13 b = np.array([[0, 10, 20], [30, 40, 50], [60, 70, 80]])
14 print(b[1:, 1:])      # [ [40 50] [70 80] ]
15 print(b[:2, :2])      # [ [ 0 10] [30 40] ]
16 print(b[::2, ::2])    # [ [ 0 20] [60 80] ]
17
18 # Etc :-)
```

# Accessing a specific column or row

To get a specific column or row in a multidimension array:

- Use an empty slice to skip a dimension
- E.g. in a 2D array, `[:,1]` gets column 1
- E.g. in a 2D array, `[1,:]` gets row 1

```
1 import numpy as np
2
3 a = np.array([[0, 10, 20], [30, 40, 50], [60, 70, 80]]
4
5 # To access a specific column ...
6 print(a[:, 0])    # [ 0 30 60]
7 print(a[:, 1])    # [10 40 70]
8 print(a[:, 2])    # [20 50 80]
9
10 # To access a specific row ...
11 print(a[0, :])   # [ 0 10 20]
12 print(a[1, :])   # [30 40 50]
13 print(a[2, :])   # [60 70 80]
14
15 # To access a specific row, simpler syntax ...
16 print(a[0])      # [ 0 10 20]
17 print(a[1])      # [30 40 50]
18 print(a[2])      # [60 70 80]
```

# Aside: Views vs. copies

When you get an array slice/row/column, you get a view on the data

- If you make any changes, it will change the actual data

If you want to get a copy of the data:

- Call `copy()` on the slice/row/column

```
1 import numpy as np
2
3 # Demonstrate views.
4 a = np.array([[0, 10, 20], [30, 40, 50], [60, 70, 80]])
5 col0View = a[:, 0]
6 col0View[2] = 600
7 print(col0View)
8 print(a) # [[ 0  10  20] [ 30  40  50] [600  70  80]]
9
10 # Demonstrate copies.
11 b = np.array([[0, 10, 20], [30, 40, 50], [60, 70, 80]])
12 col0Copy = a[:, 0].copy()
13 col0Copy[2] = 600
14 print(col0Copy)
15 print(b) # [[ 0  10  20] [ 30  40  50] [60  70  80]]
```

# Manipulating array shape

- Reshaping an array
- Creating new axes
- Concatenating arrays
- Stacking arrays vertically or horizontally
- Splitting an array

# Reshaping an array

Reshaping is a simple and common technique for creating multidimensional arrays

- Create a 1D array initially (typically)
- Reshape it to a multidimensional array (must be compatible shape)
- The multidimensional array is a view onto the original 1D array

```
1 import numpy as np
2
3 # Create 1D array initially, for simplicity.
4 a = np.arange(9)
5 print(a)                      # [0 1 2 3 4 5 6 7 8]
6 print(a.shape)                 # (9,)
7
8 # Reshape as 2D array (view on a).
9 b = a.reshape((3,3))
10 print(b)                      # [[0 1 2] [3 4 5] [6 7 8]]
11 print(b.shape)                # (3, 3)
12
13 # Changing items in b will change values in underlying
14 b[0,0] = 99
15 print(a)                      # [99 1 2 3 4 5 6 7 8]
16 print(b)                      # [[99 1 2] [ 3 4 5] [ 6 7 8]]
```

# Creating new axes

Another useful technique is create new axes for an array

- Create a 1D array initially (typically)
- Create a new column or row, using `np.newaxis`

```
1 import numpy as np
2
3 # Create 1D array initially, for simplicity.
4 a = np.arange(5)
5 print(a)          # [0 1 2 3 4]
6 print(a.shape)    # (5,)
7
8 # Create 2D array with 1 row, 5 columns.
9 b = a[np.newaxis, :]
10 print(b)         # [[0 1 2 3 4]]
11 print(b.shape)   # (1, 5)
12
13 # Create 2D array with 5 rows, 1 column.
14 c = a[:, np.newaxis]
15 print(c)         # [[0] [1] [2] [3] [4]]
16 print(c.shape)   # (5, 1)
```

# Concatenating arrays (1/2)

You can concatenate same-size arrays together

- `np.concatenate()` - you can specify the axis to concatenate on

Here's a simple example that concatenates 1D arrays

```
1 import numpy as np
2
3 # Create some 1D arrays.
4 a = np.array([ 0,  1])
5 b = np.array([10, 11])
6 c = np.array([20, 21])
7
8 # Concatenate the 1D arrays.
9 result = np.concatenate([a, b, c])
10 print(result)      # [0 1 10 11 20 21]
11 print(result.shape) # (6,)
```

# Concatenating arrays (2/2)

Here's an example that concatenates 2D arrays

- Note the optional axis parameter (default is 0)

```
1 import numpy as np
2
3 # Create some 2D arrays.
4 a = np.array([[ 0,  1], [10, 11]])
5 b = np.array([[20, 21], [30, 31]])
6 c = np.array([[40, 41], [50, 51]])
7
8 # Concatenate on axis 0 (this is the default, so can omit axis parameter).
9 result1 = np.concatenate([a, b, c], axis=0)
10 print(result1)          # [[0 1] [10 11] [20 21] [30 31] [40 41] [50 51]]
11 print(result1.shape)    # (6, 2)
12
13 # Concatenate on axis 1.
14 result2 = np.concatenate([a, b, c], axis=1)
15 print(result2)          # [[0 1 20 21 40 41] [10 11 30 31 50 51]]
16 print(result2.shape)    # (2, 6)
```

# Stacking arrays vertically or horizontally

You can stack different-size arrays together

- `np.vstack()` - stack vertically (must have same no. of cols)
- `np.hstack()` - stack horizontally (must have same no. of rows)

```
1 import numpy as np
2
3 # Create some arrays with same number of columns (2), and stack vertically.
4 a = np.array([10, 11])
5 b = np.array([[20, 21], [30, 31]])
6 result1 = np.vstack([a, b])
7 print(result1)          # [[10 11] [20 21] [30 31]]
8 print(result1.shape)    # (3, 2)
9
10 # Create some arrays with same number of rows (2), and stack horizontally.
11 c = np.array([[40, 41], [50, 51]])
12 d = np.array([[60], [61]])
13 result2 = np.hstack([c, d])
14 print(result2)          # [[40 41 60] [50 51 61]]
15 print(result2.shape)    # (2, 3)
```

# Splitting an array

You can split an array into subarrays

- `np.split()`
- `np.vsplit()`
- `np.hsplit()`

```
1 import numpy as np
2
3 # Split a 1D array.
4 a = np.arange(16)
5 a1, a2, a3, a4 = np.split(a, [2, 5, 9])
6 print('\na1\n', a1)    # [0 1]
7 print('\na2\n', a2)    # [2 3 4]
8 print('\na3\n', a3)    # [5 6 7 8]
9 print('\na4\n', a4)    # [9 10 11 12 13 14 15]
10
11 # Split a 2D vertically.
12 b = np.arange(16).reshape((4, 4))
13 b1, b2 = np.vsplit(b, [3])
14 print('\ntop\n', b1)     # [[0 1 2 3] [4 5 6 7] [8 9 1
15 print('\nbottom\n', b2)  # [[12 13 14 15]]
16
17 # Split a 2D horizontally.
18 c = np.arange(16).reshape((4, 4))
19 c1, c2 = np.hsplit(c, [3])
20 print('\nleft\n', c1)    # [[0 1 2] [4 5 6] [8 9 10] [
21 print('\nright\n', c2)   # [[3] [7] [11] [15]]
```

Any questions?

# NumPy Techniques

- NumPy universal functions
- Aggregation
- Broadcasting
- Manipulating arrays using Boolean logic
- Additional techniques

# NumPy universal functions

- Overview
- Using a loop
- Using a universal function
- Arithmetic
- Reduction and accumulation
- Additional math functions

# Overview

Universal functions are a key reason why NumPy arrays are so efficient to manipulate

A universal function is a method on a NumPy array that executes on all elements very efficiently

- Much faster and cleaner than using an explicit loop

Consider the examples on the next 2 slides...

- The 1st example uses a loop - slow and cumbersome
- The 2nd example uses a universal function - fast and elegant

You will always use universal functions rather than loops to process NumPy arrays

# Using a loop

This code uses a loop to process data in a NumPy array

- Loops through elements and applies `**` to each element
- Gathers results into another NumPy array, one-by-one

```
1 import numpy as np
2 from timeit import default_timer as timer
3
4 def compute_cubes_loop(data):
5     result = np.empty(len(data))
6     for i in range(len(data)):
7         result[i] = data[i] ** 3
8     return result
9
10 np.random.seed(0)
11 data = np.random.randint(1, 10, size=10_000_000)
12
13 start = timer()
14 cubes = compute_cubes_loop(data)
15 end = timer()
16 print('Execution time using a loop', end - start)
```

# Using a universal function

This code has the same effect, using a universal function

- Applies the `**` operator to the NumPy array itself
- NumPy applies the operator to each element implicitly

```
1 import numpy as np
2 from timeit import default_timer as timer
3
4 def compute_cubes_ufunc(data):
5     result = data ** 3
6     return result
7
8 np.random.seed(0)
9 data = np.random.randint(1, 10, size=10000000)
10
11 start = timer()
12 cubes = compute_cubes_ufunc(data)
13 end = timer()
14 print('Execution time using a universal function', end - start)
```

# Arithmetic

NumPy implements all the usual arithmetic operators as universal functions

- You can use an operator directly, or call the equivalent function

```
1 import numpy as np
2
3 a = np.arange(10)
4
5 print('Using operators')
6 print('a + 2 = ', a + 2)
7 print('a - 2 = ', a - 2)
8 print('a * 2 = ', a * 2)
9 print('a / 2 = ', a / 2)
10 print('a // 2 = ', a // 2)
11 print('a % 2 = ', a % 2)
12 print('a ** 2 = ', a ** 2)
13
14 print('\nUsing ufuncs explicitly')
15 print('np.add(a, 2)      = ', np.add(a, 2))
16 print('np.subtract(a, 2)  = ', np.subtract(a, 2))
17 print('np.multiply(a, 2)   = ', np.multiply(a, 2))
18 print('np.divide(a, 2)     = ', np.divide(a, 2))
19 print('np.floor_divide(a, 2) = ', np.floor_divide(a, 2))
20 print('np.mod(a, 2)        = ', np.mod(a, 2))
21 print('np.power(a, 2)       = ', np.power(a, 2))
```

# Reduction and accumulation

For binary ufuncs such as add, multiply, etc.

- You can call `reduce()` to reduce array to a single result
- You can call `accumulate()` to accumulate intermediate results

```
1 import numpy as np
2
3 a = np.arange(2, 5)
4
5 print('reduce() examples')
6 print('Sum    ', np.add.reduce(a))
7 print('Product', np.multiply.reduce(a))
8 print('Power   ', np.power.reduce(a))
9
10 print('accumulate() examples')
11 print('Sum    ', np.add.accumulate(a))
12 print('Product', np.multiply.accumulate(a))
13 print('Power   ', np.power.accumulate(a))
```

# Additional math functions

NumPy has universal functions for trig, log, etc.

```
1 import numpy as np
2 import scipy.special as sp # Additional specialized f
3
4 a = np.linspace(0, np.pi, 3)
5 print('sin(a) = ', np.sin(a))
6 print('cos(a) = ', np.cos(a))
7 print('tan(a) = ', np.tan(a))
8 print('sinh(a) = ', np.sinh(a))
9 print('cosh(a) = ', np.cosh(a))
10 print('tanh(a) = ', np.tanh(a))
11
12 b = np.linspace(0.1, 0.9, 3)
13 print('arcsin(b) = ', np.arcsin(b))
14 print('arccos(b) = ', np.arccos(b))
15 print('arctan(b) = ', np.arctan(b))
16 print('arcsinh(b) = ', np.arcsinh(b))
17 print('arccosh(b) = ', np.arccosh(b))
18 print('arctanh(b) = ', np.arctanh(b))
19
20 c = np.array([1, 10, 100])
21 print('log(c) = ', np.log(c))          # 2.718281 to the power of 1
22 print('log2(c) = ', np.log2(c))        # 2 to the power of 1
23 print('log10(c) = ', np.log10(c))      # 10 to the power of 1
24 print('exp(c) = ', np.exp(c))          # 2.718281 to the power of 1
25 print('exp2(c) = ', np.exp2(c))        # 2 to the power of 1
26 print('exp10(c) = ', sp.exp10(c))      # 10 to the power of 1
```

# Aggregation

- Overview
- NumPy vs. Python aggregation functions
- NumPy aggregation functions available
- Working with multidimensional arrays

# Overview

When dealing with large amounts of data, you'll probably want to compute statistics such as:

- Sum, product
- Minimum, maximum
- Mean, median, mode
- Variance, standard deviation
- Percentiles

NumPy has aggregation functions for performing these computations very efficiently

# NumPy vs. Python aggregation functions

NumPy aggregation functions look very similar to functions available in the standard Python library

- But the NumPy functions are much quicker, so use them!

Consider the following example:

```
1 import numpy as np
2 from timeit import default_timer as timer
3
4 data = np.random.rand(100_000_000)
5
6 start1 = timer()
7 result1 = sum(data)      # Python sum() function
8 end1 = timer()
9 print('Execution time using Python sum(): ', end1 - start1)
10
11 start2 = timer()
12 result2 = np.sum(data)   # NumPy sum() function
13 end2 = timer()
14 print('Execution time using NumPy sum(): ', end2 - start2)
```

# NumPy aggregation functions available

This example shows the NumPy aggregation functions

- There are also NaN-friendly functions, e.g. np.nansum()

```
1 import numpy as np
2 from scipy import stats
3
4 data = np.random.rand(100_000_000)
5
6 print('Sum', np.sum(data))
7 print('Product', np.prod(data))
8 print('Minimum', np.min(data))
9 print('Maximum', np.max(data))
10 print('Mean', np.mean(data))
11 print('Median', np.median(data))
12 print('Mode', stats.mode(data))
13 print('Variance', np.var(data))
14 print('Std dev', np.std(data))
15 print('50th percentile', np.percentile(data, 50))
```

# Working with multidimensional arrays

The aggregation functions work over the entire array

- If the array is multidimensional, all elements are processed

You can also get aggregation results for rows or columns

- Specify the axis parameter, to collapse data on that axis

```
1 import numpy as np
2
3 data = np.arange(9).reshape([3,3])
4
5 # Print the entire array.
6 print('Array data:\n', data)
7
8 # Calculate the sum over the entire array.
9 print('Sum of whole array:', np.sum(data))
10
11 # Collapse axis 0 (i.e. collapse the rows), to get sum on each column.
12 print('Sum for each column:', np.sum(data, axis=0))
13
14 # Collapse axis 1 (i.e. collapse the columns), to get sum on each row.
15 print('Sum for each row:', np.sum(data, axis=1))
```

# Broadcasting

- Universal functions and same-shape arrays
- Universal functions and different-shape arrays
- Broadcasting rules
- Understanding the broadcasting rules

# Universal functions and same-shape arrays

We discussed universal functions earlier in the chapter

- We showed how to add/subtract/etc. scalars to an array

```
1 import numpy as np
2
3 a = np.array([0, 1, 2])
4
5 result = a + 100
6 print(result)
```

Universal functions also work with arrays for both args In this example, the arrays are the same shape (3,)

```
1 import numpy as np
2
3 a1 = np.array([0, 1, 2])
4 a2 = np.array([4, 5, 6])
5
6 result = a1 + a2
7 print(result)
```

# Universal functions and different-shape arrays

Universal functions also work with different-shape arrays

- This is called broadcasting - see next slide for details

Here's a simple example of broadcasting

- a is one row, m is two rows
- The values in a are "broadcast" across both rows in m

```
1 import numpy as np
2
3 a = np.array([10, 11, 12])
4 print(a.shape)      # (3,)
5
6 m = np.array([[20, 21, 22], [30, 31, 32]])
7 print(m.shape)      # (2,3)
8
9 result = a + m
10 print(result.shape) # (2, 3)
11 print(result)       # [[30 32 34] [40 42 44]]
```

# Broadcasting rules

Broadcasting allows NumPy to stretch arrays of different shapes, to enable binary operations to take place

There are three rules about how broadcasting works, which are applied in the following order:

1. If arrays have a different number of dimensions, the shape of the array with fewer dimensions is filled with 1 on leading edge
2. If shape of arrays is different in any dimension, the array with shape 1 in that dimension is stretched to match the other shape
3. If shape in any dimension is different (and not 1), an error occurs

# Understanding the broadcasting rules

Let's re-examine the example from a couple of slides ago

```
1 a = np.array([10, 11, 12])
2 m = np.array([[20, 21, 22], [30, 31, 32]])
3 result = a + m      # [[30 32 34] [40 42 44]]
```

Let's apply broadcasting rule 1 first...

- a and m have different number of dimensions: a is 1D, m is 2D
- a has fewer dimensions, so a shape is filled with 1 on leading edge
- Thus the shape of a becomes (1, 3) i.e. 1 row of 3 columns

Now let's apply broadcasting rule 2...

- a and m have different shapes: a shape is (1,3), m shape is (2,3)
- a shape (1,3) is stretched to match m shape (2,3)

# Complex Broadcasting

In this example, both arrays need to be broadcast

```
1 import numpy as np
2
3 a = np.array([[10],[11],[12]])    # Shape (3,1)
4 b = np.array([20, 21, 22])        # Shape (3,)
5 result = a + b                  # Shape (3,3)
6 print(result)                   # [[30 31 32] [31 32 33] [32 33 34]]
```

Let's apply broadcasting rule 1 first...

- a and b have different number of dimensions: a is 2D, b is 1D
- b has fewer dimensions, so b shape is filled with 1 on leading edge
- Thus the shape of b becomes (1, 3) i.e. 1 row of 3 columns

Now let's apply broadcasting rule 2...

- a and b have different shapes: a shape is (3,1), b shape is (1,3)
- a cols stretched to match b cols, so a becomes (3,3)

# Manipulating arrays using Boolean logic

- Boolean operations
- Boolean aggregation
- Boolean masking

# Boolean operations (1/2)

We've seen how to use math operators with NumPy arrays

- `+ - \* / // etc. `

You can also use Boolean operators

- `> ≥ < ≤ = ≠ `
- Returns a NumPy array containing True/False in each position

```
1 import numpy as np
2
3 def process_marks(m):
4     print('Exam marks\n', m)
5     print('Passes?\n', m ≥ 50)
6     print('Full marks?\n', m == 100)
7     print('Not full marks?\n', m ≠ 100)
8
9 my_exam_marks = np.array([71, 95, 49, 100, 65])
10 process_marks(my_exam_marks)
11
12 our_exam_marks = np.array([[71, 95, 49, 100, 65], [99, 23, 78, 88, 92]])
13 process_marks(our_exam_marks)
```

# Boolean operations (2/2)

You can combine Boolean operations together

- & and
- | inclusive or
- ^ exclusive or
- ~ not

```
1 import numpy as np
2
3 def process_marks(m)
4     print('\nExam marks\n', m)
5     print('B?\n', (m >= 60) & (m < 70))
6     print('A or U?\n', (m >= 70) | (m < 30))
7     print('A or even, but not both?\n', (m >= 70) ^ (m % 2 == 0))
8     print('Not (A or U)?\n', ~((m >= 70) | (m < 30)))
9
10 my_exam_marks = np.array([71, 95, 49, 100, 65])
11 process_marks(my_exam_marks)
12
13 our_exam_marks = np.array([[71, 95, 49, 100, 65], [99, 22, 78, 88, 92]])
14 process_marks(our_exam_marks)
```

Note the need for parentheses for operator precedence

# Boolean aggregation

You can perform various aggregation operations on the Boolean result arrays

- `all()` - are all results True?
- `any()` - are any results True?
- `count_nonzero()` - count of non-zero (i.e. True) results

```
1 import numpy as np
2
3 def process_marks(m)
4     print('\nExam marks\n', m)
5     print('All passes? ', np.all(m ≥ 50))
6     print('Any passes? ', np.any(m ≥ 50))
7     print('Count of passes', np.count_nonzero(m ≥ 50))
8     print('Count of B    ', np.count_nonzero((m ≥ 60) & (m < 70)))
9     print('Count of A or U', np.count_nonzero((m ≥ 70) | (m < 30)))
10
11 my_exam_marks = np.array([71, 95, 49, 100, 65])
12 process_marks(my_exam_marks)
13
14 our_exam_marks = np.array([[71, 95, 49, 100, 65], [99, 22, 78, 88, 92]])
15 process_marks(our_exam_marks)
```

# Additional techniques

- Fancy indexing
- Partitioning
- Sorting

# Fancy indexing (1/3)

Often you want to get several elements at specific indices

- You can pass an array of indices into []
- Returns a result array with the elements from those indices

```
1 import numpy as np
2
3 a = np.arange(10, 20)
4
5 # Get some elements into a Python list.
6 result1 = [a[1], a[4], a[7]]
7 print('type(result1)', type(result1))
8 print('result1      ', result1)
9
10 # Get some elements into a NumPy array, using fancy ir
11 idx = [1, 4, 7]
12 result2 = a[idx]
13 print('\ntype(result2)', type(result2))
14 print('result2.shape', result2.shape)
15 print('result2      ', result2)
16
17 # Get some elements into a 2D NumPy array, using fancy
18 idx = np.array([[1, 4, 7], [2, 5, 8]])
19 result3 = a[idx]
20 print('\ntype(result3)', type(result3))
21 print('result3.shape', result3.shape)
22 print('result3      ', result3)
```

# Fancy indexing (2/3)

You can use fancy indexing with multidimensional arrays

- Specify a fancy index indicating desired rows
- Specify another fancy index indicating desired columns

```
1 import numpy as np
2
3 a = np.arange(49).reshape(7,7)
4
5 # Use fancy indexing to specify rows and columns desired.
6 ridx = [0, 2, 4]
7 cidx = [1, 3, 5]
8 result = a[ridx, cidx]
9 print('result.shape', result.shape)
10 print('result      ', result)
```

# Fancy indexing (3/3)

You can combine fancy indexing with other techniques

- E.g. regular indexing, slicing, masking

```
1 import numpy as np
2
3 a = np.arange(49).reshape(7,7)
4
5 # Combine fancy indexing with regular indexing.
6 cidx = [1, 3, 5]
7 result1 = a[2, cidx]
8 print('result1.shape', result1.shape)
9 print('result1\n',      result1)
10
11 # Combine fancy indexing with slicing.
12 cidx = [1, 3, 5]
13 result2 = a[2:5, cidx]
14 print('\nresult2.shape', result2.shape)
15 print('result2\n',      result2)
16
17 # Combine fancy indexing with masking.
18 rmask = [True, True, False, False, False, False, True]
19 cidx = [1, 3, 5]
20 result3 = a[rmask, cidx]
21 print('\nresult3.shape', result3.shape)
22 print('result3\n',      result3)
```

# Partitioning

You can partition an array via `partition()`

- You specify an index position, returns an array where all elements up to that position are smaller than all values after that position

Notes:

- Elements are unsorted within each partition
- For multidimensional arrays, the default axis of partitioning is 0
- There's also a `partition()` instance method, partitions in-place

```
1 import numpy as np
2
3 a = np.random.randint(0, 101, 12)
4 print('Unpartitioned 1D array', a)
5 print('Partitioned at index 2', np.partition(a, 2))
6 print('Partitioned at index 4', np.partition(a, 4))
7
8 b = np.random.randint(0, 101, 49).reshape((7,7))
9 print('\nUnpartitioned 2D array\n', b)
10 print('\nPartitioned at col index 2\n', b.partition(2))
11 print('\nPartitioned at col index 4\n', np.partition(b, 4))
12 print('\nPartitioned at row index 2\n', np.partition(b, 2))
13 print('\nPartitioned at row index 4\n', np.partition(b, 4))
```

# Sorting

You can sort an array via `sort()` Returns a sorted array

```
1 import numpy as np
2
3 a = np.random.randint(0, 101, 12)
4 print('Unsorted 1D array', a)
5 print('Sorted', np.sort(a))
6
7 b = np.random.randint(0, 101, 49).reshape((7,7))
8 print('\nUnsorted 2D array\n', b)
9 print('\nSorted across cols\n', np.sort(b, axis=1))
10 print('\nSorted down rows \n', np.sort(b, axis=0))
```

Notes:

- For multidimensional arrays, the default axis of partitioning is 0
- There's also a `sort()` instance method, sorts in-place
- There's also a `sort()` instance method, sorts in-place

Any questions?

# Test Automation

- Getting started with testing
- Using PyHamcrest matchers
- Testing techniques

# Getting started with testing

- Setting the scene
- Python test frameworks
- Example class-under-test
- How to write a test
- Example test
- Running tests
- Arrange / Act / Assert
- Testing for exceptions
- Setup and teardown code

# Setting the scene

Unit testing verifies the correct behaviour of your code artefacts in isolation

- In Python, a "unit" is usually a function

You typically write several unit tests per function

- To exercise all the possible paths through the function

The FIRST principles of unit testing:

- Fast
- Isolated / independent
- Repeatable
- Self-validating
- Timely

# Python test frameworks

There are several test frameworks available for Python

- Unittest (aka PyUnit) - part of standard library (OO-based)
- PyTest - simple and fast, most widely used (function-based)
- TestProject - generates HTML reports, use with PyTest or Unittest
- Behave - BDD test framework
- Robot - primarily for Acceptance Testing
- Etc.

We'll use PyTest

- Install as follows:

```
1 pip install pytest
```

Test installation as follows:

```
1 py.test -h
```

# Example class-under-test

In the next few slides we'll see how to test this simple Python class

```
1  class BankAccount:  
2  
3      def __init__(self, name):  
4          self.name = name  
5          self.balance = 0;  
6  
7      def deposit(self, amount):  
8          self.balance += amount  
9  
10     def withdraw(self, amount):  
11         if amount > self.balance:  
12             raise Exception("Insufficient funds")  
13         self.balance -= amount  
14  
15     def __str__(self):  
16         return "{} , {}".format(self.name, self.balance)  
17
```

# How to write a test

To write tests in PyTest:

- Define a separate .py file
- The filename must be `test_xxx.py` or `xxx_test.py`

Each test is a separate function

- The function name must be `test_yyy()`

Each test function should focus on a particular scenario, and should have a meaningful name

- E.g. `test_functionName_Scenario`
- E.g. `test_functionName_StateUnderTest_ExpectedBehaviour`

# Example test

When writing your tests, go for the low-hanging fruit first

- Test the simplest functions and scenarios first
- Then test the more complex functions and scenarios later

Here's a simple first test

```
1  from bankAccount import BankAccount
2
3  def test_accountCreated_zeroBalanceInitially():
4      fixture = BankAccount("David")
5      assert fixture.balance == 0
```

Notes:

- assert is a standard Python keyword
- If the test returns false, it throws an AssertionError
- This causes your test function to terminate immediately

# Running tests

To run tests in PyTest, run the following command:

```
1 py.test
```

# Arrange / Act / Assert

It's common for a test function to have 3 parts

- Arrange
- Act
- Assert

Example

```
1  def test_deposit_singleDeposit_correctBalance():
2
3      # Arrange.
4      fixture = BankAccount("David")
5
6      # Act.
7      fixture.deposit(100)
8
9      # Assert.
10     assert fixture.balance == 100
```

# Testing for exceptions (1/2)

When you write industrial-strength code, sometimes you actually want the code to throw an exception

- The code should be robust enough to detect exceptional situations
- You can write a test to verify the code throws an exception

```
1 import pytest
2 ...
3
4 def test_deposit_withdrawalsExceedLimits_exceptionOccursV1():
5
6     # Arrange.
7     fixture = BankAccount("David")
8
9     # Act.
10    fixture.deposit(600)
11
12    # Verify expected exception occurs
13    with pytest.raises(Exception):
14        fixture.withdraw(601)
15
```

# Testing for exceptions (2/2)

If you want to examine the exception that was thrown:

```
1 import pytest
2 ...
3
4 def test_deposit_withdrawalsExceedLimits_exceptionOccursV2():
5
6     # Arrange.
7     fixture = BankAccount("David")
8
9     # Act.
10    fixture.deposit(600)
11
12    # Verify expected exception occurs
13    with pytest.raises(Exception) as excinfo:
14        fixture.withdraw(601)
15
16    # Assert the exception type and error message are correct
17    assert excinfo.typename == "Exception"
18    assert excinfo.value.args[0] == "Insufficient funds"
19
```

# Setup and teardown code

Sometimes you have common setup/teardown tasks that you want to perform before/after each test

- You can define a "fixture function" to do the before/after code

```
1 import pytest
2 ...
3
4 acc = None
5
6 @pytest.fixture(autouse=True)
7 def run_around_tests():
8     # Code that will run before each test.
9     print("Do something before a test")
10    global acc
11    acc = BankAccount("David")
12
13    # A test function will be run at this point
14    yield
15
16    # Code that will run after each test.
17    print("Do something after a test")
```

Note: To see console output with PyTest, use the -s option

# Using PyHamcrest matchers

- A reminder about simple assertions
- Introducing PyHamcrest
- Getting started with PyHamcrest
- Example class-under-test
- Example test
- Defining a custom PyHamcrest matcher
- Using a custom PyHamcrest matcher

# A reminder about simple assertions

As we've seen, Python has a simple assert keyword

- `assert a == b`

What if you want to write some specific tests, such as:

- Does a collection contains a value?
- Does a variable point to a particular type of subclass?
- Does an integer value lie in a certain range?
- Does a double value equal a variable, to a specified accuracy?

# Introducing PyHamcrest

The PyHamcrest library provides a higher-level vocabulary for writing your tests, with "matcher" functions such as:

- `equal_to`, `close_to`
- `not`
- `greater_than`, `greater_than_or_equal_to`
- `less_than`, `less_than_or_equal_to`
- `starts_with`, `ends_with`, `contains_string`, `is_empty`
- `all_of`, `any_of`
- `contains_string`, `contains_exactly`, `contains_in_any_order`
- `has_item`, `has_items`
- `has_entry`, `has_entries`, `has_key`, `has_keys`, `has_value`, `has_values`
- `instance_of`
- ... etc.

# Getting started with PyHamcrest

Install PyHamcrest as follows:

```
1 pip install PyHamcrest
```

You can then use PyHamcrest as follows in your tests:

```
1 from hamcrest import *
2 ...
3
4 assert_that(... ... ...)
```

# Example class-under-test

To illustrate PyHamcrest, we'll test the following class:

```
1  class Product:  
2  
3      def __init__(self, description, price, *ratings):  
4          self.description = description  
5          self.price = price  
6          self.ratings = list(ratings)  
7  
8      def taxPayable(self):  
9          return self.price * 0.20  
10  
11     def __str__():  
12         return "{} , £{}, {}".format(self.description, self.price, self.ratings)  
13
```

# Example test

```
1  from hamcrest import *
2  import pytest
3  from Product import Product
4
5  product = None
6
7  @pytest.fixture(autouse=True)
8  def run_around_tests():
9      global product
10     product = Product("TV", 1500, 5, 4, 3, 5, 4, 3)
11     yield
12
13 def test_product_taxPayable_correct():
14     assert_that(product.taxPayable(), close_to(300, 0.1))
15
16 def test_product_ratings_containsRating():
17     assert_that(product.ratings, has_item(3))
18
19 def test_product_ratings_doesntContainsAbsentRating():
20     assert_that(product.ratings, not(has_item(2)))
```

# Defining a custom PyHamcrest matcher

The PyHamcrest "matcher" model is extensible

- You can define your own custom matcher classes

```
1  from hamcrest.core.base_matcher import BaseMatcher
2
3
4  class PriceMatcher(BaseMatcher):
5
6      def __init__(self, maxInclusive=3_000):
7          self.maxInclusive = maxInclusive
8
9      def _matches(self, price):
10         return 0 < price <= self.maxInclusive
11
12     def describe_to(self, description):
13         description.append_text("0 ... " + str(self.maxInclusive))
14
15
16     def valid_price():
17         return PriceMatcher(2_500)
```

# Using a custom PyHamcrest matcher

Here's how to use a custom PyHamcrest matcher

- Exactly the same as for the standard PyHamcrest matchers ☺

```
1  from hamcrest import *
2  from priceMatcher import valid_price
3  from product import Product
4
5
6  def test_product_validPrice_priceAccepted():
7      product1 = Product("TV", 1500)
8      assert_that(product1.price, valid_price())
9
10
11 def test_product_negativePrice_priceRejected():
12     product2 = Product("TV", -1)
13     assert_that(product2.price, is_not(valid_price()))
14
15
16 def test_product_tooExpensivePrice_priceRejected():
17     product3 = Product("TV", 2501)
18     assert_that(product3.price, is_not(valid_price()))
```

# Testing techniques

- Parameterized tests
- Running tests selectively
- Grouping tests into sets
- Test Driven Development (TDD)
- Refactoring

# Parameterized tests (1/2)

When you start writing tests, you might notice some of the tests are quite similar and repetitive

- E.g. imagine a function that returns the grade for an exam
- How would you test it always returns the correct grade?

```
1  def get_grade(mark):  
2      if mark >= 75:  
3          return "A*"  
4      elif mark >= 70:  
5          return "A"  
6      elif mark >= 60:  
7          return "B"  
8      elif mark >= 50:  
9          return "C"  
10     elif mark >= 40:  
11         return "D"  
12     elif mark >= 30:  
13         return "E"  
14     else:  
15         return "U"
```

# Parameterized tests (2/2)

You can write a parameterized test as follows:

```
1 import pytest
2 from util import get_grade
3
4 @pytest.mark.parametrize("mark,grade", [
5     (99, "A*"),
6     (70, "A"),
7     (69, "B"),
8     (60, "B"),
9     (59, "C"),
10    (50, "C"),
11    (49, "D"),
12    (40, "D"),
13    (39, "E"),
14    (30, "E"),
15    (29, "U")])
16 def test_marks_and_grades(mark, grade):
17     assert grade == get_grade(mark)
```

# Running tests selectively

test.py lets you specify which test functions to run...

E.g. run all test functions that have 'deposit' in their name

- The -k option specifies the key (function name fragment)
- The -v option displays verbose test results

```
1  py.test -k deposit -v
```

# Grouping tests into sets (1/3)

You can group tests into sets

- You can then run all the tests in a particular set

The first step is to specify your custom sets

- Define a file named pytest.ini as follows:

```
1  [pytest]
markers =
    numtest: mark a test as a numeric test
    strtest: mark a test as a string test
```

pytest.ini

# Grouping tests into sets (2/3)

You can then mark a test function so it belongs to a set(s)

- Decorate the test function with `@pytest.mark.aSetName`

```
1  import pytest
2
3
4  @pytest.mark.numtest
5  def test_add_numbers():
6      assert 3 + 4 == 7
7
8  @pytest.mark.numtest
9  def test_multiple_numbers():
10     assert 3 * 4 == 12
11
12 @pytest.mark.strtest
13 def test_concatenate_strings():
14     assert "hello " + "world" == "hello world"
15
16 @pytest.mark.strtest
17 def test_uppercase_strings():
18     assert "hello world".upper() == "HELLO WORLD"
```

# Grouping tests into sets (3/3)

To run all the tests in a particular set:

- Use the -m option
- Specifies which marked tests to run (i.e. the name of the set)

```
1  py.test -m numtest -v
```

# Test Driven Development (TDD)

TDD is a simple concept

- You write the tests first, before you write the code
- The tests act as a specification for the new functionality you're about to implement

In TDD, you perform the following tasks repeatedly:

- Write a test
- Run the test - it must fail!
- Write the minimum amount of code, to make the test pass
- Refactor your code

Benefits of TDD

- Helps you focus on functionality rather than implementation
- Ensures every line of code is tested

# Refactoring

Refactoring is an oft-overlooked aspect of TDD

- After each iteration through the test-code-pass cycle, you should refactor your code
- That is, step back and see if you can/should reorganize your code to eliminate duplication, restructure inheritance, etc.

Typical refactoring activities:

- Rename a variable / function / class / module
- Extract duplicate code into a common function
- Extract common class functionality into a superclass
- Introduce another level of inheritance

Any questions?

# GraphCore Requests

1. Spawning other processes (subprocess)
2. Regular Expressions
3. Command Line Tools

# Spawning Other Processes

- Interacting with the OS
- The subprocess module
- Using subprocess
- Capture the output
- Pass parameters
- Checking the return code
- Running another Python script

# Interacting with the OS

There are three key modules that allow us to interact with the host operating system

- The `os` module enables miscellaneous interfaces with the OS. You can use it to inspect environment variables or to get other user and process-related information.
- The `platform` module contains information about the interpreter and the machine where the process is running
- The `sys` module, which provides you with helpful system-specific information, will usually provide you with all the information that you need about the runtime environment.

# The subprocess module

- Allows us to start a new process and communicate with it
- This module has gone through some work to modernize and simplify its API and you might see code using subprocess in ways different from those shown here
- There are two main APIs: the ``subprocess.run`` which is quite high-level and ``subprocess.Popen`` which is more low level
- We'll focus on the higher level API.
- Dig into the docs if you need more complex interactions

# Using subprocess

```
1 import subprocess  
2 subprocess.run(["ls"])
```

This will run the command in your shell. If you don't have the command you'll raise an exception.

# Capture the output

```
1 import subprocess
2 result = subprocess.run(["ls"], capture_output=True)
3 print("stdout: ", result.stdout)
4 print("stderr: ", result.stderr)
5
```

Adding `text=True` will allow the `stdout` and `stderr` to be strings we can work with.

```
1 result = subprocess.run(["ls"], capture_output=True, text=True)
```

# Pass Parameters

```
1 import subprocess
2 result = subprocess.run(["ls", "-l"], capture_output=True, text=True)
3 print("stdout: ", result.stdout)
4 print("stderr: ", result.stderr)
```

Add flags and argument in the list. These will be passed through to the shell in the order they are passed.

# Checking the return code

```
1 result = subprocess.run(["ls", "non_existing_file"])
2 print("rc: ", result.returncode)
```

You can use the `returncode` attribute to decide if the process ended without error.

- 0 is success
- Anything else is some kind of failure

# Running another Python script

- You can use `sys.executable` to use the Python executable that is currently running
- We can use this to then run another script

```
1 import subprocess
2 import sys
3
4 subprocess.run([sys.executable, "subprocess2.py"])
5
```

# Regular Expressions

- Overview
- Basic Syntax
- Usage

# Overview

- A domain specific language which defines a grammar for expressing string comparisons
- Easier than having loops of checking for different things
- It's a standard language used across programming languages
- Python's implementation is from the ``re`` package in the standard library.
- Built into lots of text editors (including PyCharm and VSCode)

# Basic Syntax

- Most characters match their own identities, so "h" in a regex means "match exactly the letter h."
- Enclosing characters in square brackets can mean choosing between alternates, with "[Hh]" to mean "match either H or h."
- We use the \S character class to match all non-whitespace. Other character classes include \w (word characters), \W (non-word characters), and \d (digits).
- Two quantifiers are used:
  - ? means "0 or 1 time,"
  - The quantifier, +, means "1 or more times,"
- Parentheses () introduce a numbered sub-expression, sometimes called a "capture group."
- A backslash followed by a number refers to a numbered sub-expression, described previously. As an example, \1 refers to the first sub-expression. These can be used when replacing text that matches the regex or to store part of a regex to use later in the same expression. Because of the way that backslashes are interpreted by Python strings, this is written as `\\1` in a Python regex.

# Usage

```
1 import re
2
3 title = "And now for something completely different"
4 pattern = "(\w)\\"1+
5
6 print(re.search(pattern, title))
7 print(re.findall(pattern, title))
8
9 reObj = re.compile(pattern)
10
11 print(reObj.search(title))
```

# Another example

```
1 import re
2
3 description = "The Norwegian Blue is a wonderful parrot. This parrot is notable for its exquisite plumage."
4
5 pattern = "(parrot)"
6 replacement = "ex-\\"1"
7
8 print(re.sub(pattern, replacement, description))
9
```

# Command Line Tools

- Parse the arguments
- Carry out some work

# Add the parser

There are a number of ways to parse the arguments in Python scripts. A reliable way to do this is with ``argparse``.

First, we'll import the library:

```
1 import argparse
```

We create a parser object:

```
1 # create a parser object
2 parser = argparse.ArgumentParser(description="A description for our program")
```

# Add any arguments

Add any arguments we'd like to:

```
1 parser.add_argument("-a", "--add", nargs='*', metavar="num", type=int,  
2                     help="All the numbers separated by spaces will be added.")  
3
```

- The flags that we can pass in (normally single letter with a single dash and full word with a double). The full name will be used to reference the variables
- `nargs` - the number of arguments that come after the flag
- `metavar` - what the variable will be called in the help strings
- `type` - the type of the incoming variables
- `help` - a help string for this argument

# Parse the arguments and do some work

Once you've add the arguments, we use the `parse_args()` function.

```
1 args = parser.parse_args()
```

Then we can do some work:

```
1 if len(args.add) != 0:  
2     print(sum(args.add))  
3
```

# Labview links

I don't know enough about Labview to offer insight but here are a few resources I found which might be useful

- [https://zone.ni.com/reference/en-XX/help/371361R-01/glang/python\\_node/](https://zone.ni.com/reference/en-XX/help/371361R-01/glang/python_node/)
- [https://zone.ni.com/reference/en-XX/help/371361R-01/glang/python\\_pal/](https://zone.ni.com/reference/en-XX/help/371361R-01/glang/python_pal/)
- <https://www.youtube.com/watch?v=GApmZx-Yro>
- <https://www.youtube.com/watch?v=YL-zGx6u0sQ>
- [https://github.com/DanielleJobe/LabVIEW2018-Python-Integration-Example/tree/master/py\\_src](https://github.com/DanielleJobe/LabVIEW2018-Python-Integration-Example/tree/master/py_src)

# Further Resources

- Mastering Python Regular Expressions book - <https://www.packtpub.com/product/mastering-python-regular-expressions/9781783283156>
- Automate the Boring Stuff with Python - <https://automatetheboringstuff.com/>