



Working with Data in Python

DA2303 – Python Programming

Table of contents

- 01 Files**
Read and write files, and exception handlings
- 02 Numpy**
arrays vs lists, one dimensional and multidimensional ndarrays
- 03 Pandas**
One-dimensional Series and two-dimensional DataFrames, using csv and Pandas

01

Files

Read and write files, and
exception handlings

Files: Introduction

- Variables, lists, tuples, dictionaries, sets, arrays, pandas Series and pandas DataFrames offer only *temporary* data storage.
- The data is lost when a local variable “goes out of scope” or when the program terminates.
- **Files** provide long-term retention of typically large amounts of data, even after the program that created the data terminates, so data maintained in files is **persistent**.
- Computers store files on **secondary storage devices**, including solid-state drives, hard disks and more.

Files: Introduction (cont.)

- We consider text files in several popular formats—plain text, JSON (JavaScript Object Notation) and CSV (comma-separated values).
- We will concentrate on plain text and CSV files in this chapter.
- We'll use both the Python Standard Library's csv module and pandas to load and manipulate CSV data.

Files: Introduction (cont.)

- We will NOT COVER serializing and deserializing objects.
- You may explore on your own time using JSON to serialise and deserialize objects to facilitate saving those objects to secondary storage and transmitting them over the Internet.
- <https://docs.python.org/3/library/json.html>
- There is another library in Python Standard Library's, pickle module but there are security vulnerabilities of serializing and deserializing data with the Python Standard Library's pickle module.
- We recommend JSON serialisation in preference to pickle.

Files: Introduction (cont.)

- We also introduce **exception handling**.
- An exception indicates an execution-time problem.
- You've seen exceptions of types `ZeroDivisionError`, `NameError`, `ValueError`, `StatisticsError`, `TypeError`, `IndexError`, `KeyError` and `RuntimeError`.
- We'll show how to deal with exceptions as they occur by using `try` statements and associated `except` clauses to *handle* exceptions.
- We'll also discuss the `try` statement's `else` and `finally` clauses.

Files: Introduction (cont.)

- The features presented here help you write *robust, fault-tolerant* programs that can deal with problems and continue executing or *terminate gracefully*.
- Programs typically request and release resources (such as files) during program execution.
- Often, these are in limited supply or can be used only by one program at a time.
- We show how to guarantee that after a program uses a resource, it's released for use by other programs, even if an exception has occurred.
- You'll use the **with statement** for this purpose.

Files

- Python views a **text file** as a sequence of characters and a **binary file** (for images, videos and more) as a sequence of bytes.
- As in lists and arrays, the first character in a text file and byte in a binary file is located at position 0, so in a file of n characters or bytes, the highest position number is $n - 1$.
- The diagram below shows a conceptual view of a file:



- For each file you **open**, Python creates a **file object** that you'll use to interact with the file.

End of File

- Every operating system provides a mechanism to denote the end of a file.
- Some represent it with an **end-of-file marker** (as in the preceding figure), while others might maintain a count of the total characters or bytes in the file.
- Programming languages generally hide these operating-system details from you.

Standard File Objects

- When a Python program begins execution, it creates three **standard file objects**:
 - `sys.stdin`—the standard input file object
 - `sys.stdout`—the standard output file object, and
 - `sys.stderr`—the standard error file object.
- Though these are considered file objects, they do not read from or write to files by default.
- The `input` function implicitly uses `sys.stdin` to get user input from the keyboard.

Standard File Objects (cont.)

- Function `print` implicitly outputs to `sys.stdout`, which appears in the command line.
- Python implicitly outputs program errors and tracebacks to `sys.stderr`, which also appears in the command line.
- You must import the `sys` module if you need to refer to these objects explicitly in your code, but this is rare.

Text-File Processing

- We'll write a simple text file that might be used by an accounts-receivable system to track the money owed by a company's clients.
- We'll then read that text file to confirm that it contains the data.
- For each client, we'll store the client's account number, last name and account balance owed to the company.
- Together, these data fields represent a client **record**.
- Python imposes no structure on a file, so notions such as records do not exist natively in Python.

Text-File Processing (cont.)

- Programmers must structure files to meet their applications' requirements.
- We'll create and maintain this file in order by account number.
- In this sense, the account number may be thought of as a **record key**.

Writing to a Text File

- Let's create an accounts.txt file and write five client records to the file.
- Generally, records in text files are stored one per line, so we end each record with a newline character:

```
with open('accounts.txt', mode='w') as accounts:  
    accounts.write('100 Jones 24.98\n')  
    accounts.write('200 Doe 345.67\n')  
    accounts.write('300 White 0.00\n')  
    accounts.write('400 Stone -42.16\n')  
    accounts.write('500 Rich 224.62\n')
```



```
dj accounts.txt  
1 100 Jones 24.98  
2 200 Doe 345.67  
3 300 White 0.00  
4 400 Stone -42.16  
5 500 Rich 224.62  
6
```

Typically the file will be created in directory where you run the script.

Writing to a Text File (cont.)

- You can also write to a file with `print` (which automatically outputs a `\n`), as in

```
with open('accounts.txt', mode='w') as accounts:  
    print('100 Jones 24.98', file=accounts)  
    print('200 Doe 345.67', file=accounts)  
    print('300 White 0.00', file=accounts)  
    print('400 Stone -42.16', file=accounts)  
    print('500 Rich 224.62', file=accounts)
```



```
dj accounts.txt  
1 100 Jones 24.98  
2 200 Doe 345.67  
3 300 White 0.00  
4 400 Stone -42.16  
5 500 Rich 224.62  
6
```


The with Statement

- Many applications *acquire* resources, such as files, network connections, database connections and more.
- You should *release* resources as soon as they're no longer needed.
- This practice ensures that other applications can use the resources.
- Python's **with statement**:
 - acquires a resource (in this case, the file object for accounts.txt) and assigns its corresponding object to a variable (accounts in this example),
 - allows the application to use the resource via that variable, and
 - calls the resource object's close method to release the resource when program control reaches the end of the with statement's suite.

Built-In Function open

Be careful—
opening a file for
writing *deletes* all
the existing data
in the file.

- The built-in **open function** opens the file accounts.txt and associates it with a file object.
- The mode argument specifies the **file-open mode**, indicating whether to open a file for reading from the file, for writing to the file or both.
- The mode 'w' opens the file for *writing*, creating the file if it does not exist.
- If you do not specify a path to the file, Python creates it in the current folder where you run the script.
- By convention, the **.txt file extension** indicates a plain text file.

Writing to the File

- The with statement assigns the object returned by open to the variable accounts in the **as clause**.
- In the with statement's suite, we use the variable accounts to interact with the file.
- In this case, we call the file object's **write method** five times to write five records to the file, each as a separate line of text ending in a newline.
- At the end of the with statement's suite, the with statement *implicitly* calls the file object's **close** method to close the file.


Reading Data from a Text File

- We just created the text file `accounts.txt` and wrote data to it.
- Now let's read that data from the file sequentially from beginning to end.
- If the contents of a file should not be modified, open the file for reading only— another example of the principle of least privilege.
- This prevents the program from accidentally modifying the file.
- You open a file for reading by passing the `'r'` file-open mode as function `open`'s second argument.
- If you do not specify the folder in which to store the file, `open` assumes the file is in the current folder.

Reading Data from a Text File (cont.)

- The following session reads records from the file `accounts.txt` and displays the contents of each record in columns with the Account and Name columns *left aligned* and the Balance column *right aligned*, so the decimal points align vertically:

```
with open('accounts.txt', mode='r') as accounts:  
    print(f'{"Account":<10}{"Name":<10}{"Balance":>10}')  
    for record in accounts:  
        account, name, balance = record.split()  
        print(f'{account:<10}{name:<10}{balance:>10}')
```



Account	Name	Balance
100	Jones	24.98
200	Doe	345.67
300	White	0.00
400	Stone	-42.16
500	Rich	224.62

Reading Data from a Text File (cont.)

- Iterating through a file object, as shown in the preceding for statement, reads one line at a time from the file and returns it as a string.
- For each record (that is, line) in the file, string method split returns tokens in the line as a list, which we unpack into the variables account, name and balance.
- The last statement in the for statement's suite displays these variables in columns using field widths.

File Method readlines

- The file object's **readlines** method also can be used to read an *entire* text file.
- The method returns each line as a string in a list of strings.
- For small files, this works well, but iterating over the lines in a file object, as shown above, can be more efficient.
- Calling readlines for a large file can be a time-consuming operation, which must complete before you can begin using the list of strings.
- Using the file object in a for statement enables your program to process each text line as it's read.

Seeking to a Specific File Position

- While reading through a file, the system maintains a **file-position pointer** representing the location of the next character to read.
- Sometimes it's necessary to process a file sequentially from the beginning *several times* during a program's execution.
- Each time, you must reposition the file-position pointer to the beginning of the file, which you can do either by closing and reopening the file, or by calling the file object's **seek** method, as in

```
file_object.seek(0)
```

- The seek method approach is faster.

Updating Text Files

- Formatted data written to a text file cannot be modified without the risk of destroying other data.
- If the name 'White' needs to be changed to 'Williams' in accounts.txt, the old name cannot simply be overwritten.
- The original record for White is stored as

```
300 White 0.00
```

- If you overwrite the name 'White' with the name 'Williams', the record becomes

```
300 Williams00
```

Updating Text Files (cont.)

- The new last name contains three more characters than the original one, so the characters beyond the second “i” in 'Williams' overwrite other characters in the line.
- The problem is that in the formatted input–output model, records and their fields can vary in size.
- For example, 7, 14, −117, 2074 and 27383 are all integers and are stored in the same number of “raw data” bytes internally (typically 4 or 8 bytes in today’s systems).
- However, when these integers are output as formatted text, they become different-sized fields. For example, 7 is one character, 14 is two characters and 27383 is five characters.

Updating Text Files (cont.)

- To make the preceding name change, we can:
 - copy the records before 300 White 0.00 into a temporary file,
 - write the updated and correctly formatted record for account 300 to this file,
 - copy the records after 300 White 0.00 to the temporary file,
 - delete the old file and
 - rename the temporary file to use the original file's name.
- This can be cumbersome because it requires processing *every* record in the file, even if you need to update only one record.
- Updating a file as described above is more efficient when an application needs to update many records in one pass of the file.

Updating accounts.txt

- Let's use a with statement to update the accounts.txt file to change account 300's name from 'White' to 'Williams' as described above:

```
1  accounts = open('accounts.txt', 'r')
2  temp_file = open('temp_file.txt', 'w')
3
4  with accounts, temp_file:
5      for record in accounts:
6          account, name, balance = record.split()
7          if account != '300':
8              temp_file.write(record)
9          else:
10             new_record = ' '.join([account, 'Williams', balance])
11             temp_file.write(new_record + '\n')
```

Updating accounts.txt (cont.)

- For readability, we opened the file objects (line 1 and 2), then specified their variable names in the first line of line 3.
- This with statement manages two resource objects, specified in a comma-separated list after with.
- The for statement unpacks each record into account, name and balance.
- If the account is not '300', we write record (which contains a newline) to temp_file.
- Otherwise, we assemble the new record containing 'Williams' in place of 'White' and write it to the file.

Updating accounts.txt (cont.)

- After the end of the script, temp_file.txt contains:

dj accounts.txt

```
1 100 Jones 24.98
2 200 Doe 345.67
3 300 White 0.00
4 400 Stone -42.16
5 500 Rich 224.62
6 |
```



dj temp_file.txt

```
1 100 Jones 24.98
2 200 Doe 345.67
3 300 Williams 0.00
4 400 Stone -42.16
5 500 Rich 224.62
6 |
```

os Module File-Processing Functions

- At this point, we have the old accounts.txt file and the new temp_file.txt.
- To complete the update, let's delete the old accounts.txt file, then rename temp_file.txt as accounts.txt.
- The **os module** provides functions for interacting with the operating system, including several that manipulate your system's files and directories.

os Module File-Processing Functions (cont.)

- Now that we've created the temporary file, :

```
import os

# let's use the remove function to
# delete the original file
os.remove('accounts.txt')
# Next, let's use the rename function
# to rename the temporary file as
'accounts.txt'
os.rename('temp_file.txt', 'accounts.txt')
```

Use remove with caution—it does not warn you that you're *permanently* deleting the file. Make sure you run the script from the directory of accounts.txt that you want to delete and temp_file.txt you want to update.

Additional Notes Regarding Files

- The following table summarizes the various file-open modes for text files, including the modes for reading and writing we've introduced.
- The *writing* and *appending* modes create the file if it does not exist.
- The *reading* modes raise a `FileNotFoundError` if the file does not exist.
- Each text-file mode has a corresponding binary-file mode specified with `b`, as in `'rb'` or `'wb+'`.
- You'd use these modes, for example, if you were reading or writing binary files, such as images, audio, video, compressed ZIP files and many other popular custom file formats.

Additional Notes Regarding Files (cont.)

Mode	Description
'r'	Open a text file for reading. This is the default if you do not specify the file-open mode when you call open.
'w'	Open a text file for writing. Existing file contents are <i>deleted</i> .
'a'	Open a text file for appending at the end, creating the file if it does not exist. New data is written at the end of the file.
'r+'	Open a text file reading and writing.
'w+'	Open a text file reading and writing. Existing file contents are <i>deleted</i> .
'a+'	Open a text file reading and appending at the end. New data is written at the end of the file. If the file does not exist, it is created.

Other File Object Methods

- Here are a few more useful file-object methods.
 - For a text file, the **read** method returns a string containing the number of characters specified by the method's integer argument. For a binary file, the method returns the specified number of bytes. If no argument is specified, the method returns the entire contents of the file.
 - The **readline** method returns one line of text as a string, including the newline character if there is one. This method returns an empty string when it encounters the end of the file.
 - The **writelines** method receives a list of strings and writes its contents to a file. The classes that Python uses to create file objects are defined in the Python Standard
- Library's **io module** (<https://docs.python.org/3/library/io.html>).

Handling Exceptions

- Various types of exceptions can occur when you work with files, including:
 - A **FileNotFoundError** occurs if you attempt to open a non-existent file for reading with the 'r' or 'r+' modes.
 - A **PermissionsError** occurs if you attempt an operation for which you do not have permission. This might occur if you try to open a file that your account is not allowed to access or create a file in a folder where your account does not have permission to write, such as where your computer's operating system is stored.
 - A **ValueError** (with the error message 'I/O operation on closed file.') occurs when you attempt to write to a file that has already been closed.

Division by Zero

- In this case, the interpreter is said to **raise an exception** of type `ZeroDivisionError`. If an exception occurs in a script, Python terminates the script and displays the exception's traceback.

10 / 0



```
Traceback (most recent call last):  
  File "/Users/jailanihar/dividebyzero.py", line 1, in <module>  
    10 / 0  
ZeroDivisionError: division by zero
```

Invalid Input

- The `int` function raises a `ValueError` if you attempt to convert to an integer a string (like `'hello'`) that does not represent a number:

```
value = int(input('Enter an integer: '))
```



```
Enter an integer: hello
Traceback (most recent call last):
  File "/Users/jailanihar/test.py", line 1, in <module>
    value = int(input('Enter an integer: '))
ValueError: invalid literal for int() with base 10: 'hello'
```

try Statements

- Now let's see how to *handle* these exceptions so that you can enable code to continue processing.
- Consider the following script and sample execution.
- Its loop attempts to read two integers from the user, then display the first number divided by the second.
- The script uses exception handling to catch and handle (i.e., deal with) any `ZeroDivisionErrors` and `ValueErrors` that arise—in this case, allowing the user to re-enter the input.

try Statements (cont.)

```
1  # Simple exception handling example.
2
3  while True:
4      # attempt to convert and divide values
5      try:
6          number1 = int(input('Enter numerator: '))
7          number2 = int(input('Enter denominator: '))
8          result = number1 / number2
9      except ValueError: # tried to convert non-numeric value to int
10         print('You must enter two integers\n')
11     except ZeroDivisionError: # denominator was 0
12         print('Attempted to divide by zero\n')
13     else: # executes only if no exceptions occur
14         print(f'{number1:.3f} / {number2:.3f} = {result:.3f}')
15         break # terminate the loop
```


try Statements (cont.)

- Example input and output:

```
Enter numerator: 100
Enter denominator: 0
Attempted to divide by zero
```

```
Enter numerator: 100
Enter denominator: hello
You must enter two integers
```

```
Enter numerator: 100
Enter denominator: 7
100.000 / 7.000 = 14.286
```

try Clause

- Python uses **try statements** (like lines 5–15) to enable exception handling.
- The try statement's **try clause** (lines 5–8) begins with keyword `try`, followed by a colon (`:`) and a suite of statements that *might* raise exceptions.

except Clause

- A try clause may be followed by one or more **except clauses** (lines 9–10 and 11–12) that immediately follow the try clause's suite.
- These also are known as *exception handlers*.
- Each except clause specifies the type of exception it handles.
- In this example, each exception handler just displays a message indicating the problem that occurred.

else Clause

- After the last except clause, an optional **else clause** (lines 13–15) specifies code that should execute only if the code in the try suite did not raise exceptions.
- If no exceptions occur in this example's try suite, line 14 displays the division result and line 15 terminates the loop.

Catching Multiple Exceptions in One except Clause

- It's relatively common for a try clause to be followed by several except clauses to handle various types of exceptions.
- If several except suites are identical, you can catch those exception types by specifying them as a tuple in a *single* except handler, as in:

```
except (type1, type2, ...) as variable_name:
```

- The as clause is optional.
- Typically, programs do not need to reference the caught exception object directly.
- If you do, you can use the variable in the as clause to reference the exception object in the except suite.

What Exceptions Does a Function or Method Raise?

- Exceptions may surface via statements in a try suite, via functions or methods called directly or indirectly from a try suite, or via the Python interpreter as it executes the code (for example, `ZeroDivisionErrors`).
- Before using any function or method, read its online API documentation, which specifies what exceptions are thrown (if any) by the function or method and indicates reasons why such exceptions may occur.
- Next, read the online API documentation for each exception type to see potential reasons why such an exception occurs.

What Code Should Be Placed in a try Suite?

- Place in a try suite a significant logical section of a program in which several statements can raise exceptions, rather than wrapping a separate try statement around every statement that raises an exception.
- However, for proper exception-handling granularity, each try statement should enclose a section of code small enough that, when an exception occurs, the specific context is known and the except handlers can process the exception properly.
- If many statements in a try suite raise the same exception types, multiple try statements may be required to determine each exception's context.

finally Clause

- Operating systems typically can prevent more than one program from manipulating a file at once.
- When a program finishes processing a file, the program should close it to release the resource.
- This enables other programs to use the file (if they're allowed to access it).
- Closing the file helps prevent a **resource leak** in which the file resource is not available to other programs because a program using the file never closes it.

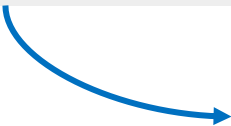
The finally Clause of the try Statement

- A try statement may have a finally clause as its last clause after any except clauses or else clause.
- The **finally** clause is guaranteed to execute, *regardless* of whether its try suite executes successfully or an exception occurs.
- In other languages that have finally, this makes the finally suite an ideal location to place resource-deallocation code for resources acquired in the corresponding try suite.
- In Python, we prefer the with statement for this purpose and place other kinds of “clean up” code in the finally suite.

The finally Clause of the try Statement (cont.)

- The following demonstrates that the finally clause always executes, regardless of whether an exception occurs in the corresponding try suite. First, let's consider a try statement in which no exceptions occur in the try suite:

```
try:  
    print('try suite with no exceptions raised')  
except:  
    print('this will not execute')  
else:  
    print('else executes because no exceptions in the try suite')  
finally:  
    print('finally always executes')
```



try suite with no exceptions raised
else executes because no exceptions in the try suite
finally always executes


The finally Clause of the try Statement (cont.)

- The preceding try suite displays a message but does not raise any exceptions.
- When program control successfully reaches the end of the try suite, the except clause is skipped, the else clause executes and the finally clause displays a message showing that it always executes.
- When the finally clause terminates, program control continues with the next statement after the try statement.

The finally Clause of the try Statement (cont.)

- Now let's consider a try statement in which an exception occurs in the try suite:

```
try:
    print('try suite that raises an exception')
    int('hello')
    print('this will not execute')
except ValueError:
    print('a ValueError occurred')
else:
    print('else will not execute because an exception occurred')
finally:
    print('finally always executes')
```



try suite that raises an exception
a ValueError occurred
finally always executes

The finally Clause of the try Statement (cont.)

- This try suite begins by displaying a message.
- The second statement attempts to convert the string 'hello' to an integer, which causes the int function to raise a ValueError.
- The try suite immediately terminates, skipping its last print statement.
- The except clause catches the ValueError exception and displays a message.
- The else clause does not execute because an exception occurred

The finally Clause of the try Statement (cont.)

- Then, the finally clause displays a message showing that it always executes.
- When the finally clause terminates, program control continues with the next statement after the try statement.

Combining with Statements and try...except Statements

- Most resources that require explicit release, such as files, network connections and data- base connections, have potential exceptions associated with processing those resources.
- For example, a program that processes a file might raise IOErrors. For this reason, *robust* file- processing code normally appears in a try suite containing a with statement to guarantee that the resource gets released.
- The code is in a try suite, so you can catch in except handlers any exceptions that occur and you do not need a finally clause because the with statement handles resource deallocation.

Combining with Statements and try...except Statements (cont.)

- To demonstrate this, first let's assume you're asking the user to supply the name of a file and they provide that name incorrectly, such as gradez.txt rather than the file we created earlier grades.txt.
- In this case, the open call raises a FileNotFoundError by attempting to open a non-existent file.
- To catch exceptions like FileNotFoundError that occur when you try to open a file for reading, wrap the with statement in a try suite.

Combining with Statements and try...except Statements (cont.)

```
try:
    with open('gradez.txt', 'r') as accounts:
        print(f'{"ID":<3}{"Name":<7}{"Grade"}')
        for record in accounts:
            student_id, name, grade = record.split()
            print(f'{student_id:<3}{name:<7}{grade}')
except FileNotFoundError:
    print('The file name you specified does not exist')
```



The file name you specified does not exist

Explicitly Raising an Exception

- You've seen various exceptions raised by your Python code.
- Sometimes you might need to write functions that raise exceptions to inform callers of errors that occur.
- The **raise** statement explicitly raises an exception. The simplest form of the raise statement is

```
raise ExceptionClassName
```

- The raise statement creates an object of the specified exception class.

Explicitly Raising an Exception

- Optionally, the exception class name may be followed by parentheses containing arguments to initialize the exception object—typically to provide a custom error message string.
- Code that raises an exception first should release any resources acquired before the exception occurred.
- In most cases, when you need to raise an exception, it's recommended that you use one of Python's many built-in exception types listed at:
- <https://docs.python.org/3/library/exceptions.html>

02

Numpy

arrays vs lists, one
dimensional and
multidimensional
ndarrays

NumPy

- The NumPy (Numerical Python) library first appeared in 2006 and is the preferred Python array implementation.
- It offers a high-performance, richly functional n-dimensional array type called ndarray, which from this point forward we'll refer to by its synonym, array.
- NumPy is one of the many open-source libraries that the Anaconda Python distribution installs.

NumPy (cont.)

- Operations on arrays are up to two orders of magnitude faster than those on lists.
- In a big-data world in which applications may do massive amounts of processing on vast amounts of array-based data, this performance advantage can be critical.
- According to libraries.io, over 450 Python libraries depend on NumPy.
- Many popular data science libraries such as Pandas, SciPy (Scientific Python) and Keras (for deep learning) are built on or depend on NumPy.

NumPy (cont.)

- Lists can have multiple dimensions.
- You generally process multi-dimensional lists with nested loops or list comprehensions with multiple for clauses.
- A strength of NumPy is “array-oriented programming,” which uses functional-style programming with internal iteration to make array manipulations concise and straightforward, eliminating the kinds of bugs that can occur with the external iteration of explicitly programmed loops.

Installing NumPy

- You can install NumPy using conda command.

```
conda install numpy
```


How to Import NumPy

- To access NumPy and its functions import it in your Python code like this:

```
import numpy as np
```

- We shorten the imported name to np for better readability of code using NumPy.
- This is a widely adopted convention that you should follow so that anyone working with your code can easily understand it.

Example: NumPy

- The numpy module provides various functions for creating arrays. Here we use the array function, which receives as an argument an array or other collection of elements and returns a new array containing the argument's elements. Let's pass a list:

```
import numpy as np
```

```
numbers = np.array([2, 3, 5, 7, 11])  
print(type(numbers))  
print(numbers)
```



```
<class 'numpy.ndarray'>  
[ 2  3  5  7 11]
```

Example: NumPy – How it works


- Note that the type is `numpy.ndarray`, but all arrays are output as “array.”
- When outputting an array, NumPy separates each value from the next with a comma and a space and right-aligns all the values using the same field width.
- It determines the field width based on the value that occupies the largest number of character positions.
- In this case, the value 11 occupies the two character positions, so all the values are formatted in two-character fields.
- That’s why there’s a leading space between the [and 2.

Multidimensional Arguments

- The array function copies its argument's dimensions. Let's create an array from a two- row-by-three-column list:

```
import numpy as np
```

```
numbers = np.array([[1, 2, 3], [4, 5, 6]])  
print(numbers)
```



```
[[1 2 3]  
 [4 5 6]]
```


- NumPy auto-formats arrays, based on their number of dimensions, aligning the columns within each row.

array Attributes

- An array object provides attributes that enable you to discover information about its structure and contents. In this section we'll use the following arrays:

```
import numpy as np

integers = np.array([[1, 2, 3], [4, 5, 6]])
print(integers)
floats = np.array([0.0, 0.1, 0.2, 0.3, 0.4])
print(floats)
```



```
[[1 2 3]
 [4 5 6]]
[0.  0.1 0.2 0.3 0.4]
```


- NumPy does not display trailing 0s to the right of the decimal point in floating-point values.

Determining an array's Element Type

- The array function determines an array's element type from its argument's elements. You can check the element type with an array's dtype attribute:

```
import numpy as np
```

```
integers = np.array([[1, 2, 3], [4, 5, 6]])  
print(integers.dtype)  
floats = np.array([0.0, 0.1, 0.2, 0.3, 0.4])  
print(floats.dtype)
```



```
int64  
float64
```

int32 and float32
on some
platforms

- As you'll see in the next section, various array-creation functions receive a dtype keyword argument so you can specify an array's element type.

Determining an array's Element Type (cont.)


- For performance reasons, NumPy is written in the C programming language and uses C's data types.
- By default, NumPy stores integers as the NumPy type `int64` values—which correspond to 64-bit (8-byte) integers in C—and stores floating-point numbers as the NumPy type `float64` values—which correspond to 64-bit (8-byte) floating-point values in C.
- In our examples, most commonly you'll see the types `int64`, `float64`, `bool` (for Boolean) and `object` for non-numeric data (such as strings).
- The complete list of supported types is at <https://docs.scipy.org/doc/numpy/user/basics.types.html>

Determining an array's Dimensions

- The attribute `ndim` contains an array's number of dimensions and the attribute `shape` contains a tuple specifying an array's dimensions:

```
import numpy as np

integers = np.array([[1, 2, 3], [4, 5, 6]])
print(integers.ndim)
print(integers.shape)
floats = np.array([0.0, 0.1, 0.2, 0.3, 0.4])
print(floats.ndim)
print(floats.shape)
```



```
2
(2, 3)
1
(5,)
```


- Here, `integers` has 2 rows and 3 columns (6 elements) and `floats` is one-dimensional, so `floats` shows a one-element tuple (indicated by the comma) containing `floats`' number of elements (5).

Determining an array's Number of Elements and Element Size

- You can view an array's total number of elements with the attribute `size` and the number of bytes required to store each element with `itemsize`:

```
import numpy as np

integers = np.array([[1, 2, 3], [4, 5, 6]])
print(integers.size)
print(integers.itemsize)
floats = np.array([0.0, 0.1, 0.2, 0.3, 0.4])
print(floats.size)
print(floats.itemsize)
```



6
8
5
8


Note that integers' size is the product of the shape tuple's values—two rows of three elements each for a total of six elements. In each case, `itemsize` is 8 because integers contains `int64` values and floats contains `float64` values, which each occupy 8 bytes.

Iterating Through a Multidimensional array's Elements

- You'll generally manipulate arrays using concise functional-style programming techniques. However, because arrays are iterable, you can use external iteration if you'd like:

```
import numpy as np

integers = np.array([[1, 2, 3], [4, 5, 6]])
for row in integers:
    for column in row:
        print(column, end=' ')
    print()
```




1	2	3
4	5	6

Iterating Through a Multidimensional array's Elements (cont.)

- You can iterate through a multidimensional array as if it were one-dimensional by using its flat attribute:

```
import numpy as np

integers = np.array([[1, 2, 3], [4, 5, 6]])
for i in integers.flat:
    print(i, end=' ')
print()
```



1 2 3 4 5 6

Filling arrays with Specific Values


- NumPy provides functions `zeros`, `ones` and `full` for creating arrays containing 0s, 1s or a specified value, respectively.
- By default, `zeros` and `ones` create arrays containing float64 values.
- We'll show how to customize the element type momentarily.
- The first argument to these functions must be an integer or a tuple of integers specifying the desired dimensions.
- For an integer, each function returns a one-dimensional array with the specified number of elements:

Filling arrays with Specific Values

- For an integer, each function returns a one-dimensional array with the specified number of elements:

```
import numpy as np
```

```
floats = np.zeros(5)  
print(floats)
```




```
[0. 0. 0. 0. 0.]
```

- For a tuple of integers, these functions return a multidimensional array with the specified dimensions. You can specify the array's element type with the zeros and ones function's dtype keyword argument:

```
import numpy as np
```

```
integers = np.ones((2, 4), dtype=int)  
print(integers)
```




```
[[1 1 1 1]  
 [1 1 1 1]]
```

Filling arrays with Specific Values (cont.)

- The array returned by full contains elements with the second argument's value and type:

```
import numpy as np
```

```
integers = np.full((3, 5), 13)  
print(integers)
```



```
[[13 13 13 13 13]  
 [13 13 13 13 13]  
 [13 13 13 13 13]]
```

Creating Integer Ranges with arange

Though you can create arrays by passing ranges as arguments, always use `arange` as it's optimized for arrays.

- NumPy provides optimized functions for creating arrays from ranges. We focus on simple evenly spaced integer and floating-point ranges, but NumPy also supports nonlinear ranges.
- `arange` first determines the resulting array's number of elements, allocates the memory, then stores the specified range of values in the array:

```
import numpy as np  
  
print(np.arange(5))  
print(np.arange(5, 10))  
print(np.arange(10, 1, -2))
```




```
[0 1 2 3 4]  
[5 6 7 8 9]  
[10 8 6 4 2]
```

Creating Floating-Point Ranges with linspace

- You can produce evenly spaced floating-point ranges with NumPy's `linspace` function.
- The function's first two arguments specify the starting and ending values in the range, and the ending value is included in the array. The optional keyword argument `num` specifies the number of evenly spaced values to produce—this argument's default value is 50:

```
import numpy as np
```

```
print(np.linspace(0.0, 1.0, num=5))
```




```
[0.  0.25 0.5  0.75 1.  ]
```


Reshaping an array

- You also can create an array from a range of elements, then use array method reshape to transform the one-dimensional array into a multidimensional array. Let's create an array containing the values from 1 through 20, then reshape it into four rows by five columns:

```
import numpy as np

integers = np.arange(1, 21)
print(integers)
reshape_integers = integers.reshape(4, 5)
print(reshape_integers)
```



```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
```


You can reshape any array, provided that the new shape has the same number of elements as the original. So a six-element one-dimensional array can become a 3-by-2 or 2-by-3 array, and vice versa, but attempting to reshape a 15-element array into a 4-by-4 array (16 elements) causes a `ValueError`.

Displaying Large arrays

- When displaying an array, if there are 1000 items or more, NumPy drops the middle rows, columns or both from the output.
- The following generate 100,000 elements. The first case shows all four rows but only the first and last three of the 25,000 columns.
- The notation ... represents the missing data. The second case shows the first and last three of the 100 rows, and the first and last three of the 1000 columns:

```
import numpy as np
```

```
print(np.arange(1, 100001).reshape(4, 25000))  
print(np.arange(1, 100001).reshape(100, 1000))
```



[[1	2	3	...	24998	24999	25000]
[25001	25002	25003	...	49998	49999	50000]	
[50001	50002	50003	...	74998	74999	75000]	
[75001	75002	75003	...	99998	99999	100000]	
[[1	2	3	...	998	999	1000]
[1001	1002	1003	...	1998	1999	2000]	
[2001	2002	2003	...	2998	2999	3000]	
...								
[97001	97002	97003	...	97998	97999	98000]	
[98001	98002	98003	...	98998	98999	99000]	
[99001	99002	99003	...	99998	99999	100000]	


Arithmetic Operations with arrays and Individual Numeric Values

- NumPy provides many operators which enable you to write simple expressions that perform operations on entire arrays. Here, we demonstrate arithmetic between arrays and numeric values and between arrays of the same shape.
- First, let's perform element-wise arithmetic with arrays and numeric values by using arithmetic operators and augmented assignments.
- Element-wise operations are applied to every element

Arithmetic Operations with arrays and Individual Numeric Values (cont.)

```
import numpy as np

numbers = np.arange(1, 6)
print(numbers)
# multiplies every element by 2
print(numbers * 2)
# cubes every element
print(numbers ** 3)
# numbers is unchanged by the arithmetic operators
print(numbers)
# This will modify every element in the left operand
numbers += 10
print(numbers)
```




```
[1 2 3 4 5]
[ 2  4  6  8 10]
[  1  8 27 64 125]
[1 2 3 4 5]
[11 12 13 14 15]
```

Broadcasting

- Normally, the arithmetic operations require as operands two arrays of the same size and shape.
- When one operand is a single value, called a scalar, NumPy performs the element-wise calculations as if the scalar were an array of the same shape as the other operand, but with the scalar value in all its elements.
- This is called broadcasting. Previous example each use this capability.

```
import numpy as np

numbers = np.arange(1, 6)
print(numbers)
# multiplies every element by 2
print(numbers * 2)
# is equivalent to
print(numbers * [2, 2, 2, 2, 2])
```




```
[1 2 3 4 5]
[ 2  4  6  8 10]
[ 2  4  6  8 10]
```

Arithmetic Operations Between arrays

- You may perform arithmetic operations and augmented assignments between arrays of the same shape. Let's multiply the one-dimensional arrays `numbers` and `numbers2` that each contain five elements:

```
import numpy as np
```

```
numbers = np.arange(1, 6)  
numbers2 = np.linspace(1.1, 5.5, 5)  
print(numbers)  
print(numbers2)  
print(numbers * numbers2)
```



```
[1 2 3 4 5]  
[1.1 2.2 3.3 4.4 5.5]  
[ 1.1  4.4  9.9 17.6 27.5]
```

The result is a new array formed by multiplying the arrays element-wise in each operand
 $11 * 1.1, 12 * 2.2, 13 * 3.3$, etc.


Arithmetic between arrays of integers and floating-point numbers results in an array of floating-point numbers.

Comparing arrays

- You can compare arrays with individual values and with other arrays.
- Comparisons are performed element-wise. Such comparisons produce arrays of Boolean values in which each element's True or False value indicates the comparison result:

```
import numpy as np
```

```
numbers = np.arange(11, 16)  
numbers2 = np.linspace(1.1, 5.5, 5)  
print(numbers)  
print(numbers2)  
print(numbers >= 13)  
print(numbers2 < numbers)  
print(numbers == numbers2)  
print(numbers == numbers)
```



```
[11 12 13 14 15]  
[1.1 2.2 3.3 4.4 5.5]  
[False False  True  True  True]  
[ True  True  True  True  True]  
[False False False False False]  
[ True  True  True  True  True]
```

NumPy Calculation Methods

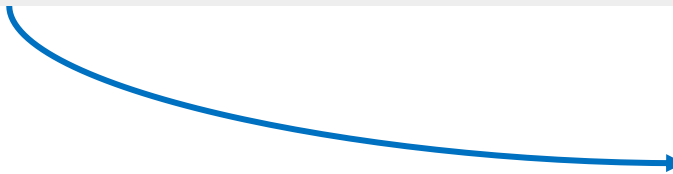
- An array has various methods that perform calculations using its contents.
- By default, these methods ignore the array's shape and use all the elements in the calculations.
- For example, calculating the mean of an array totals all of its elements regardless of its shape, then divides by the total number of elements.
- You can perform these calculations on each dimension as well.
- For example, in a two-dimensional array, you can calculate each row's mean and each column's mean.

NumPy Calculation Methods

- Consider an array representing four students' grades on three exams:

```
import numpy as np

grades = np.array([[87, 96, 70], [100, 87, 90], [94, 77, 90], [100, 81, 82]])
print(grades.sum())
print(grades.min())
print(grades.max())
print(grades.mean())
print(grades.std())
print(grades.var())
```



1054
70
100
87.83333333333333
8.792357792739987
77.30555555555556

Calculations by Row or Column

- Many calculation methods can be performed on specific array dimensions, known as the array's axes.
- These methods receive an axis keyword argument that specifies which dimension to use in the calculation, giving you a quick way to perform calculations by row or column in a two-dimensional array.
- Assume that you want to calculate the average grade on each exam, represented by the columns of grades.

Calculations by Row or Column (cont.)

- Specifying axis=0 performs the calculation on all the row values within each column:

```
import numpy as np

grades = np.array([[87, 96, 70], [100, 87, 90], [94, 77, 90], [100, 81, 82]])
print(grades.mean(axis=0))
```



```
[95.25 85.25 83.  ]
```

- So 95.25 above is the average of the first column's grades (87, 100, 94 and 100), 85.25 is the average of the second column's grades (96, 87, 77 and 81) and 83 is the average of the third column's grades (70, 90, 90 and 82).

Calculations by Row or Column (cont.)

NumPy arrays have many more calculation methods. For the complete list, see https://docs.scipy.org/doc/numpy/reference/array_s.ndarray.html

- Similarly, specifying `axis=1` performs the calculation on all the column values within each individual row. To calculate each student's average grade for all exams, we can use:

```
import numpy as np

grades = np.array([[87, 96, 70], [100, 87, 90], [94, 77, 90], [100, 81, 82]])
print(grades.mean(axis=1))
```



```
[84.33333333 92.33333333 87.66666667]
```

- This produces four averages—one each for the values in each row. So 84.33333333 is the average of row 0's grades (87, 96 and 70), and the other averages are for the remaining rows.


Universal Functions

The expression
`np.add(numbers, numbers2)`
is equivalent to:
`numbers + numbers2`

- NumPy offers dozens of standalone universal functions (or ufuncs) that perform various element-wise operations.
- Each performs its task using one or two array or array-like (such as lists) arguments.
- Some of these functions are called when you use operators like `+` and `*` on arrays. Each returns a new array containing the results.

```
import numpy as np

numbers = np.array([1, 4, 9, 16, 25, 36])
print(numbers)
print(np.sqrt(numbers))
numbers2 = np.arange(1, 7) * 10
print(numbers2)
print(np.add(numbers, numbers2))
```



```
[ 1  4  9 16 25 36]
[1.  2.  3.  4.  5.  6.]
[10 20 30 40 50 60]
[11 24 39 56 75 96]
```

Broadcasting with Universal Functions

- Let's use the multiply universal function to multiply every element of numbers2 by the scalar value 5:

```
import numpy as np

numbers2 = np.arange(1, 7) * 10
print(numbers2)
# is equivalent to: numbers2 * 5
print(np.multiply(numbers2, 5))
```



```
[10 20 30 40 50 60]
[ 50 100 150 200 250 300]
```


You can view the broadcasting rules at:
<https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html>

Broadcasting with Universal Functions (cont.)

- Let's reshape numbers2 into a 2-by-3 array, then multiply its values by a one-dimensional array of three elements:

```
import numpy as np

numbers2 = np.arange(1, 7) * 10
print(numbers2)
numbers3 = numbers2.reshape(2, 3)
print(numbers3)
numbers4 = np.array([2, 4, 6])
print(numbers4)
print(np.multiply(numbers3, numbers4))
# This works because numbers4 has the
# same length as each row of numbers3
```



```
[10 20 30 40 50 60]
[[10 20 30]
 [40 50 60]]
[2 4 6]
[[ 20  80 180]
 [ 80 200 360]]
```

If a universal function receives two arrays of different shapes that do not support broadcasting, a `ValueError` occurs.

Other Universal Functions

- The NumPy documentation lists universal functions in five categories—math, trigonometry, bit manipulation, comparison and floating point. The following table lists some functions from each category. You can view the complete list, their descriptions and more information about universal functions at:

NumPy universal functions

Math—add, subtract, multiply, divide, remainder, exp, log, sqrt, power, and more.

Trigonometry—sin, cos, tan, hypot, arcsin, arccos, arctan, and more.

Bit manipulation—bitwise_and, bitwise_or, bitwise_xor, invert, left_shift and right_shift.

Comparison—greater, greater_equal, less, less_equal, equal, not_equal, logical_and, logical_or, logical_xor, logical_not, minimum, maximum, and more.

Floating point—floor, ceil, isinf, isnan, fabs, trunc, and more.


Indexing with Two-Dimensional arrays

- One-dimensional arrays can be indexed and sliced using the same syntax and techniques we demonstrated in the Sequences. Here, we focus on array-specific indexing and slicing capabilities.
- To select an element in a two-dimensional array, specify a tuple containing the element's row and column indices in square brackets

```
import numpy as np

grades = np.array([[87, 96, 70], [100, 87, 90],
                  [94, 77, 90], [100, 81, 82]])

print(grades)
print(grades[0, 1]) # row 0, column 1
```



```
[[ 87  96  70]
 [100  87  90]
 [ 94  77  90]
 [100  81  82]]
96
```

Selecting a Subset of a Two-Dimensional array's Rows

```
import numpy as np


grades = np.array([[87, 96, 70], [100, 87, 90],
                  [94, 77, 90], [100, 81, 82]])

print(grades)

# To select a single row,
# specify only one index in square brackets:
print(grades[1])

# To select multiple sequential rows,
# use slice notation:
print(grades[0:2])

# To select multiple non-sequential rows,
# use a list of row indices:
print(grades[[1, 3]])
```



```
[[ 87  96  70]
 [100  87  90]
 [ 94  77  90]
 [100  81  82]]

[100  87  90]

[[ 87  96  70]
 [100  87  90]]

[[100  87  90]
 [100  81  82]]
```


Selecting a Subset of a Two-Dimensional array's Columns

- You can select subsets of the columns by providing a tuple specifying the row(s) and column(s) to select. Each can be a specific index, a slice or a list. Let's select only the elements in the first column:

```
import numpy as np

grades = np.array([[87, 96, 70], [100, 87, 90],
                  [94, 77, 90], [100, 81, 82]])

print(grades)
print(grades[:, 0])
```



```
[[ 87  96  70]
 [100  87  90]
 [ 94  77  90]
 [100  81  82]]
[ 87 100  94 100]
```


- The 0 after the comma indicates that we're selecting only column 0. The : before the comma indicates which rows within that column to select. In this case, : is a slice representing all rows.

Selecting a Subset of a Two-Dimensional array's Columns (cont.)

```
import numpy as np

grades = np.array([[87, 96, 70], [100, 87, 90],
                  [94, 77, 90], [100, 81, 82]])

print(grades)
# You can select consecutive columns using a slice:
print(grades[:, 1:3])
# or specific columns using a list of column indices:
print(grades[:, [0, 2]])
```



```
[[ 87  96  70]
 [100  87  90]
 [ 94  77  90]
 [100  81  82]]

[[96 70]
 [87 90]
 [77 90]
 [81 82]]

[[ 87  70]
 [100  90]
 [ 94  90]
 [100  82]]
```

Views: Shallow Copies

- Views are also known as shallow copies.
- Various array methods and slicing operations produce views of an array's data.
- The array method `view` returns a new array object with a view of the original array object's data.

Views: Shallow Copies (cont.)

- First, let's create an array and a view of that array:


```
import numpy as np
```

```
numbers = np.arange(1, 6)
```

```
numbers2 = numbers.view()
```

```
print(numbers)
```

```
print(numbers2)
```




```
[1 2 3 4 5]  
[1 2 3 4 5]
```

Views: Shallow Copies (cont.)

- We can use the built-in `id` function to see that `numbers` and `numbers2` are different objects:

```
import numpy as np

numbers = np.arange(1, 6)
numbers2 = numbers.view()
print(numbers)
print(numbers2)
print(id(numbers))
print(id(numbers2))
```




```
[1 2 3 4 5]
[1 2 3 4 5]
140413622587728
140413622587920
```

Views: Shallow Copies (cont.)

- To prove that numbers2 views the same data as numbers, let's modify an element in numbers, then display both arrays (Similarly, changing a value in the view also changes that value in the original array)

```
import numpy as np

numbers = np.arange(1, 6)
numbers2 = numbers.view()
print(numbers)
print(numbers2)
numbers[1] *= 10
print(numbers)
print(numbers2)
numbers2[1] /= 10
print(numbers)
print(numbers2)
```




```
[1 2 3 4 5]
[1 2 3 4 5]
[ 1 20 3 4 5]
[ 1 20 3 4 5]
[1 2 3 4 5]
[1 2 3 4 5]
```


Slice Views

- Slices also create views. Let's make numbers2 a slice that views only the first three elements of numbers:

```
import numpy as np
```

```
numbers = np.arange(1, 6)  
numbers2 = numbers[0:3]  
print(numbers)  
print(numbers2)  
print(id(numbers))  
print(id(numbers2))
```



```
[1 2 3 4 5]  
[1 2 3]  
140465832296784  
140465832296976
```

Slice Views (cont.)

- We can confirm that numbers2 is a view of only the first three numbers elements by attempting to access numbers2[3], which produces an IndexError:

```
import numpy as np

numbers = np.arange(1, 6)
numbers2 = numbers[0:3]
print(numbers)
print(numbers2)
print(numbers[3])
print(numbers2[3])
```

[1 2 3 4 5]

[1 2 3]

4

Traceback (most recent call last):

File "/Users/jailanihar/sliceview.py", line 8, in <module>

print(numbers2[3])


IndexError: index 3 is out of bounds for axis 0 with size 3

Slice Views (cont.)

- Now, let's modify an element both arrays share, then display them. Again, we see that numbers2 is a view of numbers:

```
import numpy as np

numbers = np.arange(1, 6)
numbers2 = numbers[0:3]
print(numbers)
print(numbers2)
numbers[1] *= 20
print(numbers)
print(numbers2)
```



```
[1 2 3 4 5]
[1 2 3]
[ 1 40 3 4 5]
[ 1 40 3]
```

Deep Copies


- Though views are separate array objects, they save memory by sharing element data from other arrays.
- However, when sharing mutable values, sometimes it's necessary to create a deep copy with independent copies of the original data.
- This is especially important in multi-core programming, where separate parts of your program could attempt to modify your data at the same time, possibly corrupting it.

Deep Copies (cont.)

- The array method `copy` returns a new array object with a deep copy of the original array object's data. First, let's create an array and a deep copy of that array:

```
import numpy as np
```

```
numbers = np.arange(1, 6)  
numbers2 = numbers.copy()  
print(numbers)  
print(numbers2)  
numbers[1] *= 10  
print(numbers)  
print(numbers2)
```



```
[1 2 3 4 5]  
[1 2 3 4 5]  
[ 1 20 3 4 5]  
[1 2 3 4 5]
```

reshape vs. resize


- We've used array method reshape to produce two-dimensional arrays from one-dimensional ranges.
- NumPy provides various other ways to reshape arrays.
- The array methods reshape and resize both enable you to change an array's dimensions.
- Method reshape returns a *view* (shallow copy) of the original array with the new dimensions. It does *not* modify the original array.
- Method resize modifies the original array's shape.

reshape vs. resize (cont.)

```
import numpy as np

grades = np.array([[87, 96, 70],
                  [100, 87, 90]])

print(grades)
grades.reshape(1, 6)
print(grades)
grades.resize(1, 6)
print(grades)
```



```
[[ 87  96  70]
 [100  87  90]]
[[ 87  96  70]
 [100  87  90]]
[[ 87  96  70 100  87  90]]
```


flatten vs. ravel

- You can take a multidimensional array and flatten it into a single dimension with the methods **flatten** and **ravel**. Method **flatten** *deep copies* the original array's data:

```
import numpy as np

grades = np.array([[87, 96, 70],
                  [100, 87, 90]])

print(grades)
flattened = grades.flatten()
print(flattened)
print(grades)
flattened[0] = 100
print(flattened)
print(grades)
```



```
[[ 87  96  70]
 [100  87  90]]
[ 87  96  70 100  87  90]
[[ 87  96  70]
 [100  87  90]]
[100  96  70 100  87  90]
[[ 87  96  70]
 [100  87  90]]
```



flatten vs. ravel (cont.)

- Method `ravel` produces a *view* of the original array, which *shares* the grades array's data:

```
import numpy as np

grades = np.array([[87, 96, 70],
                  [100, 87, 90]])

print(grades)
raveled = grades.ravel()
print(raveled)
print(grades)
raveled[0] = 100
print(raveled)
print(grades)
```



[[87 96 70]
 [100 87 90]]

[87 96 70 100 87 90]

[[87 96 70]
 [100 87 90]]

[100 96 70 100 87 90]

[[100 96 70]
 [100 87 90]]

Transposing Rows and Columns

- You can quickly **transpose** an array's rows and columns—that is “flip” the array, so the rows become the columns and the columns become the rows.
- The **T attribute** returns a transposed *view* (shallow copy) of the array.

Transposing Rows and Columns (cont.)

- The original grades array represents two students' grades (the rows) on three exams (the columns).
- Let's transpose the rows and columns to view the data as the grades on three exams (the rows) for two students (the columns):

```
import numpy as np

grades = np.array([[87, 96, 70],
                  [100, 87, 90]])

print(grades)
print(grades.T)
print(grades)
```



```
[[ 87  96  70]
 [100  87  90]]
[[ 87 100]
 [ 96  87]
 [ 70  90]]
[[ 87  96  70]
 [100  87  90]]
```


Horizontal and Vertical Stacking

- You can combine arrays by adding more columns or more rows—known as *horizontal stacking* and *vertical stacking*.
- Let's create another 2-by-3 array of grades.
- Let's assume grades2 represents three additional exam grades for the two students in the grades array.

```
import numpy as np

grades = np.array([[87, 96, 70],
                  [100, 87, 90]])
grades2 = np.array([[94, 77, 90],
                   [100, 81, 82]])

print(grades)
print(grades2)
```



```
[[ 87  96  70]
 [100  87  90]]
[[ 94  77  90]
 [100  81  82]]
```


Horizontal and Vertical Stacking (cont.)

- Let's assume `grades2` represents three additional exam grades for the two students in the `grades` array. We can combine `grades` and `grades2` with NumPy's **`hstack` (horizontal stack) function** by passing a tuple containing the arrays to combine. The extra parentheses are required because `hstack` expects one argument:

```
import numpy as np

grades = np.array([[87, 96, 70],
                  [100, 87, 90]])
grades2 = np.array([[94, 77, 90],
                   [100, 81, 82]])

print(grades)
print(grades2)
print(np.hstack((grades, grades2)))
```



```
[[ 87  96  70]
 [100  87  90]
 [ 94  77  90]
 [100  81  82]
 [ 87  96  70  94  77  90]
 [100  87  90 100  81  82]]
```


Horizontal and Vertical Stacking (cont.)

- Next, let's assume that `grades2` represents two more students' grades on three exams. In this case, we can combine `grades` and `grades2` with NumPy's **`vstack`** (**vertical stack**) function:

```
import numpy as np

grades = np.array([[87, 96, 70],
                  [100, 87, 90]])
grades2 = np.array([[94, 77, 90],
                   [100, 81, 82]])

print(grades)
print(grades2)
print(np.vstack((grades, grades2)))
```



```
[[ 87  96  70]
 [100  87  90]
 [ 94  77  90]
 [100  81  82]
 [ 87  96  70]
 [100  87  90]
 [ 94  77  90]
 [100  81  82]]
```

Learn more about NumPy

- There are more thing you can do in NumPy.
- You may refer to the documentation at <https://numpy.org/doc/stable/user/index.html>

03

Pandas

One-dimensional Series
and two-dimensional
DataFrames, using csv
and Pandas

Pandas

- NumPy's array is optimized for homogeneous numeric data that's accessed via integer indices.
- Data science presents unique demands for which more customized data structures are required.
- Big data applications must support mixed data types, customized indexing, missing data, data that's not structured consistently and data that needs to be manipulated into forms appropriate for the databases and data analysis packages you use.
- **Pandas** is the most popular library for dealing with such data.

Pandas (cont.)

- It provides two key collections that you'll use—**Series** for one-dimensional collections and **DataFrames** for two-dimensional collections.
- You can use pandas' **MultIndex** to manipulate multi-dimensional data in the context of Series and DataFrames.
- NumPy and pandas are intimately related. Series and DataFrames use arrays “under the hood.”
- Series and DataFrames are valid arguments to many NumPy operations.
- Similarly, arrays are valid arguments to many Series and DataFrame operations.

Pandas (cont.)

- Pandas is a massive topic—the PDF of its documentation is over 2000 pages.
- In this module, we present an introduction to pandas.
- We discuss its Series and DataFrames collections, and use them in support of data preparation.
- You'll see that Series and DataFrames make it easy for you to perform common tasks like selecting elements a variety of ways, filter/map/reduce operations (central to functional-style programming and big data), mathematical operations, visualization and more.

pandas: Data Structure

Dimensions	Name	Description
1	Series	1D labeled homogeneously-typed array
2	DataFrame	General 2D labeled, size- mutable tabular structure with potentially heterogeneously-typed column

- The best way to think about the pandas data structures is as flexible containers for lower dimensional data.
- For example, DataFrame is a container for Series, and Series is a container for scalars.

Installing pandas

- You can install pandas using conda command.

```
conda install pandas
```

How to Import pandas

- To load the pandas package and start working with it, import the package:

```
import pandas as pd
```

- The community agreed alias for pandas is pd, so loading pandas as pd is assumed standard practice for all of the pandas documentation.

Series

- A **Series** is an enhanced one-dimensional array.
- Whereas arrays use only zero-based integer indices, Series support custom indexing, including even non-integer indices like strings.
- Series also offer additional capabilities that make them more convenient for many data-science oriented tasks.
- For example, Series may have missing data, and many Series operations ignore missing data by default.

Creating a Series with Default Indices

- By default, a Series has integer indices numbered sequentially from 0. The following creates a Series of student grades from a list of integers:

```
import pandas as pd  
  
grades = pd.Series([87, 100, 94])
```


- The initializer also may be a tuple, a dictionary, an array, another Series or a single value.
- We'll show a single value momentarily.

Displaying a Series

- Pandas displays a Series in two-column format with the indices *left aligned* in the left column and the values *right aligned* in the right column.
- After listing the Series elements, pandas shows the data type (dtype) of the underlying array's elements:

```
import pandas as pd
```

```
grades = pd.Series([87, 100, 94])  
print(grades)
```



```
0    87  
1   100  
2    94  
dtype: int64
```


- Note how easy it is to display a Series in this format, compared to the corresponding code for displaying a list in the same two-column format.

Creating a Series with All Elements Having the Same Value

- You can create a series of elements that all have the same value:

```
import pandas as pd
```

```
grades = pd.Series(98.6, range(3))  
print(grades)
```



```
0    98.6  
1    98.6  
2    98.6  
dtype: float64
```

- The second argument is a one-dimensional iterable object (such as a list, an array or a range) containing the Series' indices. The number of indices determines the number of elements.

Accessing a Series' Elements

- You can access a Series's elements by via square brackets containing an index:

```
import pandas as pd
```

```
grades = pd.Series([87, 100, 94])
```

```
print(grades[0])
```



87


Producing Descriptive Statistics for a Series

- Series provides many methods for common tasks including producing various descriptive statistics.
- Here we show count, mean, min, max and std (standard deviation):

Each of these is a functional-style reduction.

```
import pandas as pd

grades = pd.Series([87, 100, 94])
print(grades.count())
print(grades.mean())
print(grades.min())
print(grades.max())
print(grades.std())
```




```
3
93.66666666666667
87
100
6.506407098647712
```

Producing Descriptive Statistics for a Series (cont.)

- Calling Series method **describe** produces all these stats and more:

```
import pandas as pd

grades = pd.Series([87, 100, 94])
print(grades.describe())
```



count	3.000000
mean	93.666667
std	6.506407
min	87.000000
25%	90.500000
50%	94.000000
75%	97.000000
max	100.000000
dtype:	float64

- The 25%, 50% and 75% are **quartiles**:
 - 50% represents the median of the sorted values.
 - 25% represents the median of the first half of the sorted values.
 - 75% represents the median of the second half of the sorted values.


Producing Descriptive Statistics for a Series (cont.)

- For the quartiles, if there are two middle elements, then their average is that quartile's median.
- We have only three values in our Series, so the 25% quartile is the average of 87 and 94, and the 75% quartile is the average of 94 and 100.
- Together, the **interquartile range** is the 75% quartile minus the 25% quartile, which is another measure of dispersion, like standard deviation and variance.
- Of course, quartiles and interquartile range are more useful in larger datasets.

Creating a Series with Custom Indices

- You can specify *custom* indices with the index keyword argument:

```
import pandas as pd  
  
grades = pd.Series([87, 100, 94],  
                   index=['Wally', 'Eva', 'Sam'])  
print(grades)
```



Wally	87
Eva	100
Sam	94
dtype: int64	

- In this case, we used string indices, but you can use other immutable types, including integers not beginning at 0 and nonconsecutive integers.
- Again, notice how nicely and concisely pandas formats a Series for display.


Dictionary Initialisers

- If you initialise a Series with a dictionary, its keys become the Series' indices, and its values become the Series' element values:

```
import pandas as pd

grades = pd.Series({'Wally': 87,
                   'Eva': 100,
                   'Sam': 94})

print(grades)
```



Wally	87
Eva	100
Sam	94
dtype:	int64

Accessing Elements of a Series Via Custom Indices

- In a Series with custom indices, you can access individual elements via square brackets containing a custom index value:

```
import pandas as pd

grades = pd.Series({'Wally': 87,
                   'Eva': 100,
                   'Sam': 94})

print(grades['Eva'])
```



100

Accessing Elements of a Series Via Custom Indices (cont.)

- If the custom indices are strings that could represent valid Python identifiers, pandas automatically adds them to the Series as attributes that you can access via a dot (.), as in:

```
import pandas as pd

grades = pd.Series({'Wally': 87,
                   'Eva': 100,
                   'Sam': 94})

print(grades.Wally)
```



87

Accessing Elements of a Series Via Custom Indices (cont.)

- Series also has *built-in* attributes. For example, the **dtype** attribute returns the underlying array's element type:

```
import pandas as pd

grades = pd.Series({'Wally': 87,
                   'Eva': 100,
                   'Sam': 94})

print(grades.dtype)
```



int64

Accessing Elements of a Series Via Custom Indices (cont.)

- and the **values** attribute returns the underlying array:

```
import pandas as pd

grades = pd.Series({'Wally': 87,
                   'Eva': 100,
                   'Sam': 94})

print(grades.values)
```




```
[ 87 100  94]
```

Creating a Series of Strings

- If a Series contains strings, you can use its **str attribute** to call string methods on the elements.
- First, let's create a Series of hardware-related strings:

```
import pandas as pd  
  
hardware = pd.Series(['Hammer', 'Saw', 'Wrench'])  
print(hardware)
```



```
0    Hammer  
1      Saw  
2    Wrench  
dtype: object
```


- Note that pandas also *right-aligns* string element values and that the dtype for strings is object.

Creating a Series of Strings (cont.)

- Let's call string method `contains` on each element to determine whether the value of each element contains a lowercase 'a':

```
import pandas as pd

hardware = pd.Series(['Hammer', 'Saw', 'Wrench'])
print(hardware.str.contains('a'))
```




0	True
1	True
2	False
dtype: bool	

- Pandas returns a Series containing bool values indicating the `contains` method's result for each element—the element at index 2 ('Wrench') does not contain an 'a', so its element in the resulting Series is False.
- The `str` attribute provides many string-processing methods that are similar to those in Python's string type. For a list, see: <https://pandas.pydata.org/pandas-docs/stable/api.html#string-handling>.

Creating a Series of Strings (cont.)

- The following uses string method upper to produce a *new* Series containing the uppercase versions of each element in hardware:

```
import pandas as pd  
  
hardware = pd.Series(['Hammer', 'Saw', 'Wrench'])  
print(hardware.str.upper())
```



```
0    HAMMER  
1     SAW  
2   WRENCH  
dtype: object
```

DataFrames

- A **DataFrame** is an enhanced two-dimensional array.
- Like Series, DataFrames can have custom row and column indices, and offer additional operations and capabilities that make them more convenient for many data-science oriented tasks.
- DataFrames also support missing data.
- Each column in a DataFrame is a Series.
- The Series representing each column may contain different element types, as you'll soon see when we discuss loading datasets into DataFrames.


Creating a DataFrame from a Dictionary

- Let's create a DataFrame from a dictionary that represents student grades on three exams:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
print(grades)
```



	Wally	Eva	Sam	Katie	Bob
0	87	100	94	100	83
1	96	87	77	81	65
2	70	90	90	82	85

Creating a DataFrame from a Dictionary (cont.)

- Pandas displays DataFrames in tabular format with the indices *left aligned* in the index column and the remaining columns' values *right aligned*.
- The dictionary's *keys* become the column names and the *values* associated with each key become the element values in the corresponding column.
- Shortly, we'll show how to “flip” the rows and columns.
- By default, the row indices are auto-generated integers starting from 0.

Customizing a DataFrame's Indices with the index Attribute


- We could have specified custom indices with the index keyword argument when we created the DataFrame

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict,
                      index=['Test1',
                             'Test2',
                             'Test3'])

print(grades)
```



	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test2	96	87	77	81	65
Test3	70	90	90	82	85


Customizing a DataFrame's Indices with the index Attribute (cont.)

- Let's use the **index attribute** to change the DataFrame's indices from sequential integers to labels:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades)
```



	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test2	96	87	77	81	65
Test3	70	90	90	82	85

Customizing a DataFrame's Indices with the index Attribute (cont.)

- When specifying the indices, you must provide a one-dimensional collection that has the same number of elements as there are *rows* in the DataFrame; otherwise, a `ValueError` occurs.
- Series also provides an **index attribute** for changing an existing Series' indices.


Accessing a DataFrame's Columns

- One benefit of pandas is that you can quickly and conveniently look at your data in many different ways, including selecting portions of the data.
- Let's start by getting Eva's grades by name, which displays her column as a Series:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades['Eva'])
```



Test1	100
Test2	87
Test3	90

Name: Eva, dtype: int64


Accessing a DataFrame's Columns (cont.)

- If a DataFrame's column-name strings are valid Python identifiers, you can use them as attributes.
- Let's get Sam's grades with the *Sam attribute*:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.Sam)
```



Test1	94
Test2	77
Test3	90

Name: Sam, dtype: int64

Selecting Rows via the loc and iloc Attributes

- Though DataFrames support indexing capabilities with [], the pandas documentation recommends using the attributes loc, iloc, at and iat, which are optimized to access DataFrames and also provide additional capabilities beyond what you can do only with [].
- Also, the documentation states that indexing with [] *often* produces a copy of the data, which is a logic error if you attempt to assign new values to the DataFrame by assigning to the result of the [] operation.


Selecting Rows via the loc and iloc Attributes (cont.)

- You can access a row by its label via the DataFrame's **loc** attribute. The following lists all the grades in the row 'Test1':

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.loc['Test1'])
```



Wally	87
Eva	100
Sam	94
Katie	100
Bob	83
Name: Test1, dtype: int64	


Selecting Rows via the loc and iloc Attributes (cont.)

- You also can access rows by integer zero-based indices using the **iloc** attribute (the i in iloc means that it's used with integer indices). The following lists all the grades in the second row:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.iloc[1])
```



Wally	96
Eva	87
Sam	77
Katie	81
Bob	65
Name: Test2, dtype: int64	


Selecting Rows via Slices and Lists with the loc and iloc Attributes

- The index can be a *slice*. When using slices containing labels with loc, the range specified *includes* the high index ('Test3'):

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.loc['Test1':'Test3'])
```



	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test2	96	87	77	81	65
Test3	70	90	90	82	85


Selecting Rows via Slices and Lists with the loc and iloc Attributes (cont.)

- When using slices containing integer indices with iloc, the range you specify *excludes* the high index (2):

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.iloc[0:2])
```



	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test2	96	87	77	81	65


Selecting Rows via Slices and Lists with the loc and iloc Attributes (cont.)

- To select *specific rows*, use a *list* rather than slice notation with loc or iloc:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.loc[['Test1', 'Test3']])
print(grades.iloc[[0, 2]])
```



	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test3	70	90	90	82	85

	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test3	70	90	90	82	85

Selecting Subsets of the Rows and Columns

- So far, we've selected only *entire* rows.
- You can focus on small subsets of a DataFrame by selecting rows *and* columns using two slices, two lists or a combination of slices and lists.
- Suppose you want to view only Eva's and Katie's grades on Test1 and Test2.

Selecting Subsets of the Rows and Columns

- We can do that by using loc with a slice for the two consecutive rows and a list for the two non-consecutive columns:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.loc['Test1':'Test2',
                 ['Eva', 'Katie']])
```



	Eva	Katie
Test1	100	100
Test2	87	81

Selecting Subsets of the Rows and Columns (cont.)

- The slice 'Test1':'Test2' selects the rows for Test1 and Test2.
- The list ['Eva', 'Katie'] selects only the corresponding grades from those two columns.


Selecting Subsets of the Rows and Columns (cont.)

- Let's use `iloc` with a list and a slice to select the first and third tests and the first three columns for those tests:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.iloc[[0, 2], 0:3])
```



	Wally	Eva	Sam
Test1	87	100	94
Test3	70	90	90


Boolean Indexing

- One of pandas' more powerful selection capabilities is **Boolean indexing**.
- For example, let's select all the A grades—that is, those that are greater than or equal to 90:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades[grades >= 90])
```



	Wally	Eva	Sam	Katie	Bob
Test1	NaN	100.0	94.0	100.0	NaN
Test2	96.0	NaN	NaN	NaN	NaN
Test3	NaN	90.0	90.0	NaN	NaN

Boolean Indexing (cont.)

- Pandas checks every grade to determine whether its value is greater than or equal to 90 and, if so, includes it in the new DataFrame.
- Grades for which the condition is False are represented as **NaN (not a number)** in the new DataFrame.
- NaN is pandas' notation for missing values.


Boolean Indexing (cont.)

- Let's select all the B grades in the range 80–89:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades[(grades >= 80)
             & (grades < 90)])
```



	Wally	Eva	Sam	Katie	Bob
Test1	87.0	NaN	NaN	NaN	83.0
Test2	NaN	87.0	NaN	81.0	NaN
Test3	NaN	NaN	NaN	82.0	85.0

Boolean Indexing (cont.)

- Pandas Boolean indices combine multiple conditions with the Python operator `&` (bitwise AND), *not* the `and` Boolean operator.
- For or conditions, use `|` (bitwise OR).
- NumPy also supports Boolean indexing for arrays, but always returns a one-dimensional array containing only the values that satisfy the condition.

Accessing a Specific DataFrame Cell by Row and Column

- You can use a DataFrame's **at** and **iat** attributes to get a single value from a DataFrame.
- Like `loc` and `iloc`, `at` uses labels and `iat` uses integer indices.
- In each case, the row and column indices must be separated by a comma.

Accessing a Specific DataFrame Cell by Row and Column (cont.)

- Let's select Eva's Test2 grade (87) and Wally's Test3 grade (70)

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.at['Test2', 'Eva'])
print(grades.iat[2, 0])
```



87
70

Accessing a Specific DataFrame Cell by Row and Column (cont.)

- You also can assign new values to specific elements. Let's change Eva's Test2 grade to 100 using `at`, then change it back to 87 using `iat`:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.at['Test2', 'Eva'])
grades.at['Test2', 'Eva'] = 100
print(grades.at['Test2', 'Eva'])
print(grades.iat[1, 2])
grades.iat[1, 2] = 87
print(grades.iat[1, 2])
```



87
100
77
87

Descriptive Statistics

- Both Series and DataFrames have a **describe method** that calculates basic descriptive statistics for the data and returns them as a DataFrame.
- In a DataFrame, the statistics are calculated by column (again, soon you'll see how to flip rows and columns):

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.describe())
```

Descriptive Statistics (cont.)

- The output

	Wally	Eva	Sam	Katie	Bob
count	3.000000	3.000000	3.000000	3.000000	3.000000
mean	84.333333	92.333333	87.000000	87.666667	77.666667
std	13.203535	6.806859	8.888194	10.692677	11.015141
min	70.000000	87.000000	77.000000	81.000000	65.000000
25%	78.500000	88.500000	83.500000	81.500000	74.000000
50%	87.000000	90.000000	90.000000	82.000000	83.000000
75%	91.500000	95.000000	92.000000	91.000000	84.000000
max	96.000000	100.000000	94.000000	100.000000	85.000000

Descriptive Statistics (cont.)

- As you can see, describe gives you a quick way to summarize your data.
- Pandas handles internally all the details of calculating these statistics for each column.
- You might be interested in seeing similar statistics on test-by-test basis so you can see how all the students performs on Tests 1, 2 and 3—we'll show how to do that shortly.


Descriptive Statistics (cont.)

- By default, pandas calculates the descriptive statistics with floating-point values and displays them with six digits of precision.
- You can control the precision and other default settings with pandas' `set_option` function:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
pd.set_option('precision', 2)
print(grades.describe())
```



	Wally	Eva	Sam	Katie	Bob
count	3.00	3.00	3.00	3.00	3.00
mean	84.33	92.33	87.00	87.67	77.67
std	13.20	6.81	8.89	10.69	11.02
min	70.00	87.00	77.00	81.00	65.00
25%	78.50	88.50	83.50	81.50	74.00
50%	87.00	90.00	90.00	82.00	83.00
75%	91.50	95.00	92.00	91.00	84.00
max	96.00	100.00	94.00	100.00	85.00


Descriptive Statistics (cont.)

- For student grades, the most important of these statistics is probably the mean.
- You can calculate that for each student simply by calling mean on the DataFrame:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
pd.set_option('precision', 2)
print(grades.mean())
```



Wally	84.33
Eva	92.33
Sam	87.00
Katie	87.67
Bob	77.67
dtype:	float64

Transposing the DataFrame with the T Attribute


- You can quickly **transpose** the rows and columns—so the rows become the columns, and the columns become the rows—by using the **T** attribute:

T returns a transposed *view* (not a copy) of the DataFrame.

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.T)
```



	Test1	Test2	Test3
Wally	87	96	70
Eva	100	87	90
Sam	94	77	90
Katie	100	81	82
Bob	83	65	85


Transposing the DataFrame with the T Attribute

- Let's assume that rather than getting the summary statistics by student, you want to get them by test. Simply call `describe` on `grades.T`, as in:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.T.describe())
```



	Test1	Test2	Test3
count	5.000000	5.000000	5.000000
mean	92.800000	81.200000	83.400000
std	7.661593	11.54123	8.234076
min	83.000000	65.000000	70.000000
25%	87.000000	77.000000	82.000000
50%	94.000000	81.000000	85.000000
75%	100.000000	87.000000	90.000000
max	100.000000	96.000000	90.000000


Transposing the DataFrame with the T Attribute

- To see the average of all the students' grades on each test, just call mean on the T attribute:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.T.mean())
```



Test1	92.8
Test2	81.2
Test3	83.4
dtype:	float64

Sorting by Rows by Their Indices

- You'll often sort data for easier readability.
- You can sort a DataFrame by its rows or columns, based on their indices or values.
- Let's sort the rows by their *indices* in *descending* order using **sort_index** and its keyword argument `ascending=False` (the default is to sort in *ascending* order)

Sorting by Rows by Their Indices

- This returns a new DataFrame containing the sorted data:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.sort_index(ascending=False))
```



	Wally	Eva	Sam	Katie	Bob
Test3	70	90	90	82	85
Test2	96	87	77	81	65
Test1	87	100	94	100	83

Sorting by Column Indices

- Now let's sort the columns into ascending order (left-to-right) by their column names.
- Passing the **axis=1 keyword argument** indicates that we wish to sort the *column* indices, rather than the row indices—axis=0 (the default) sorts the *row* indices:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.sort_index(axis=1))
```



	Bob	Eva	Katie	Sam	Wally
Test1	83	100	100	94	87
Test2	65	87	81	77	96
Test3	85	90	82	90	70


Sorting by Column Values

- Let's assume we want to see Test1's grades in descending order so we can see the students' names in highest-to-lowest grade order. We can call the method **sort_values** as follows:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.sort_values(by='Test1',
                        axis=1, ascending=False))
```



	Eva	Katie	Sam	Wally	Bob
Test1	100	100	94	87	83
Test2	87	81	77	96	65
Test3	90	82	90	70	85

Sorting by Column Values (cont.)

- The `by` and `axis` keyword arguments work together to determine which values will be sorted.
- In this case, we sort based on the column values (`axis=1`) for `Test1`.


Sorting by Column Values (cont.)

- Of course, it might be easier to read the grades and names if they were in a column, so we can sort the transposed DataFrame instead.
- Here, we did not need to specify the axis keyword argument, because `sort_values` sorts data in a specified column by default:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.T.sort_values(by='Test1',
                           ascending=False))
```



	Test1	Test2	Test3
Eva	100	87	90
Katie	100	81	82
Sam	94	77	90
Wally	87	96	70
Bob	83	65	85


Sorting by Column Values (cont.)

- Finally, since you're sorting only Test1's grades, you might not want to see the other tests at all. So, let's combine selection with sorting:

```
import pandas as pd

grades_dict = {'Wally': [87, 96, 70],
               'Eva': [100, 87, 90],
               'Sam': [94, 77, 90],
               'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}

grades = pd.DataFrame(grades_dict)
grades.index = ['Test1', 'Test2', 'Test3']
print(grades.loc['Test1']
      .sort_values(ascending=False))
```



Eva	100
Katie	100
Sam	94
Wally	87
Bob	83

Name: Test1, dtype: int64

Copy vs. In-Place Sorting

- By default the `sort_index` and `sort_values` return a *copy* of the original DataFrame, which could require substantial memory in a big data application.
- You can sort the DataFrame *in place*, rather than *copying* the data.
- To do so, pass the keyword argument `inplace=True` to either `sort_index` or `sort_values`.

Working with CSV Files

- **CSV (comma-separated values)** is a particularly popular file format.
- In this section, we'll demonstrate CSV file processing with a Python Standard Library module and pandas.

Python Standard Library Module

CSV

- The **csv module** provides functions for working with CSV files.
- Many other Python libraries also have built-in CSV support.

Writing to a CSV File

- Let's create an accounts.csv file using CSV format.
- The csv module's documentation recommends opening CSV files with the additional keyword argument `newline=''` to ensure that newlines are processed properly:

```
import csv

with open('accounts.csv', mode='w', newline='') as accounts:
    writer = csv.writer(accounts)
    writer.writerow([100, 'Jones', 24.98])
    writer.writerow([200, 'Doe', 345.67])
    writer.writerow([300, 'White', 0.00])
    writer.writerow([400, 'Stone', -42.16])
    writer.writerow([500, 'Rich', 224.62])
```

Writing to a CSV File (cont.)

- The **.csv file extension** indicates a CSV-format file.
- The csv module's **writer function** returns an object that writes CSV data to the specified file object.
- Each call to the writer's **writerow method** receives an iterable to store in the file.
- Here we're using lists. By default, writerow delimits values with commas, but you can specify custom delimiters.
- After the preceding snippet, accounts.csv contains:

```
accounts.csv
1 100,Jones,24.98
2 200,Doe,345.67
3 300,White,0.0
4 400,Stone,-42.16
5 500,Rich,224.62
6
```

Writing to a CSV File (cont.)

- CSV files generally do not contain spaces after commas, but some people use them to enhance readability.
- The `writerow` calls above can be replaced with one **`writerows`** call that outputs a comma-separated list of iterables representing the records.
- If you write data that contains commas within a given string, `writerow` encloses that string in double quotes.
- For example, consider the following Python list:

```
[100, 'Jones, Sue', 24.98]
```

Writing to a CSV File (cont.)

- The single-quoted string 'Jones, Sue' contains a comma separating the last name and first name. In this case, `writerow` would output the record as

```
100,"Jones, Sue",24.98
```


- The quotes around "Jones, Sue" indicate that this is a *single* value.
- Programs reading this from a CSV file would break the record into *three* pieces—100, 'Jones, Sue' and 24.98.

Reading from a CSV File

- Now let's read the CSV data from the file.
- The following reads records from the file accounts.csv and displays the contents of each record, producing the same output we showed earlier:

```
import csv

with open('accounts.csv', 'r', newline='') as accounts:
    print(f'{"Account":<10}{"Name":<10}{"Balance":>10}')
    reader = csv.reader(accounts)
    for record in reader:
        account, name, balance = record
        print(f'{account:<10}{name:<10}{balance:>10}')
```



Account	Name	Balance
100	Jones	24.98
200	Doe	345.67
300	White	0.0
400	Stone	-42.16
500	Rich	224.62

Reading from a CSV File (cont.)

- The csv module's **reader function** returns an object that reads CSV-format data from the specified file object.
- Just as you can iterate through a file object, you can iterate through the reader object one record of comma-delimited values at a time.
- The preceding for statement returns each record as a list of values, which we unpack into the variables account, name and balance, then display.

Caution: Commas in CSV Data Fields

- Be careful when working with strings containing embedded commas, such as the name 'Jones, Sue'.
- If you accidentally enter this as the two strings 'Jones' and 'Sue', then `writerow` would, of course, create a CSV record with *four* fields, not *three*.
- Programs that read CSV files typically expect every record to have the *same* number of fields; otherwise, problems occur.

Caution: Commas in CSV Data Fields

- For example, consider the following two lists:

```
[100, 'Jones', 'Sue', 24.98]  
[200, 'Doe', 345.67]
```

- The first list contains *four* values and the second contains only *three*.
- If these two records were written into the CSV file, then read into a program using the previous snippet, the following statement would fail when we attempt to unpack the four-field record into only three variables:

```
account, name, balance = record
```

Caution: Missing Commas and Extra Commas in CSV Files

- Be careful when preparing and processing CSV files.
- For example, suppose your file is composed of records, each with *four* comma-separated int values, such as:

```
100,85,77,9
```

- If you accidentally omit one of these commas, as in:

```
100,8577,9
```

- then the record has only *three* fields, one with the invalid value 8577.

Caution: Missing Commas and Extra Commas in CSV Files (cont.)

- If you put two adjacent commas where only one is expected, as in:

```
100,85,,77,9
```

- then you have *five* fields rather than *four*, and one of the fields erroneously would be *empty*.
- Each of these comma-related errors could confuse programs trying to process the record.

Reading CSV Files into Pandas DataFrames

- We covered many Pandas fundamentals.
- Here, we demonstrate pandas' ability to load files in CSV format, then perform some basic data-analysis tasks.


Working with Locally Stored CSV Files

- You can load a CSV dataset into a DataFrame with the pandas function `read_csv`.
- The following loads and displays the CSV file `accounts.csv` that you created earlier in this chapter:

```
import pandas as pd

df = pd.read_csv('accounts.csv',
                 names=['account',
                       'name',
                       'balance'])

print(df)
```



	account	name	balance
0	100	Jones	24.98
1	200	Doe	345.67
2	300	White	0.00
3	400	Stone	-42.16
4	500	Rich	224.62


Working with Locally Stored CSV Files (cont.)

- The names keyword argument specifies the DataFrame's column names.
- If you do not supply this argument, read_csv assumes that the CSV file's first row is a comma-delimited list of column names.
- To save a DataFrame to a file using CSV format, call DataFrame method to_csv:

```
import pandas as pd

df = pd.read_csv('accounts.csv',
                 names=['account',
                       'name',
                       'balance'])

df.to_csv('accounts_from_dataframe.csv', index=False)
```



accounts_from_dataframe.csv

1	account,name,balance
2	100,Jones,24.98
3	200,Doe,345.67
4	300,White,0.0
5	400,Stone,-42.16
6	500,Rich,224.62
7	

Working with Locally Stored CSV Files (cont.)

- The `index=False` keyword argument indicates that the row names (0–4 at the left of the DataFrame's output) are not written to the file.
- The resulting file contains the column names as the first row:

 accounts_from_dataframe.csv

```
1  account,name,balance
2  100,Jones,24.98
3  200,Doe,345.67
4  300,White,0.0
5  400,Stone,-42.16
6  500,Rich,224.62
7  |
```


Reading the Titanic Disaster Dataset

- The Titanic disaster dataset is one of the most popular machine-learning datasets.
- In Machine Learning, it is possible to use this dataset to “predict” whether passengers would live or die, based *only* on attributes like gender, age and passenger class.
- But we will NOT be doing that in this module.
- The dataset is available in many formats, including CSV.

Loading the Titanic Dataset via a URL

- If you have a URL representing a CSV dataset, you can load it into a DataFrame with `read_csv`.
- Let's load the Titanic Disaster dataset directly from GitHub:

```
import pandas as pd

titanic = pd.read_csv('https://vincentarelbundock.github.io/' +
                      'Rdatasets/csv/carData/TitanicSurvival.csv')
```

Viewing Some of the Rows in the Titanic Dataset

- This dataset contains over 1300 rows, each representing one passenger.
- According to Wikipedia, there were approximately 1317 passengers and 815 of them died.
- For large datasets, displaying the DataFrame shows only the first 30 rows, followed by “...” and the last 30 rows.
- To save space, let's view the first five and last five rows with DataFrame methods **head** and **tail**.

Viewing Some of the Rows in the Titanic Dataset (cont.)

- Both methods return five rows by default, but you can specify the number of rows to display as an argument:

```
import pandas as pd

titanic = pd.read_csv('https://vincentarelbundock.github.io/' +
                      'Rdatasets/csv/carData/TitanicSurvival.csv')
pd.set_option('precision', 2)
print(titanic.head())
print(titanic.tail())
```

Viewing Some of the Rows in the Titanic Dataset (cont.)

- Output

```
      Unnamed: 0 survived    sex    age passengerClass
0  Allen, Miss. Elisabeth Walton    yes  female    29.00         1st
1  Allison, Master. Hudson Trevor    yes   male     0.92         1st
2    Allison, Miss. Helen Loraine    no  female     2.00         1st
3  Allison, Mr. Hudson Joshua Crei    no   male    30.00         1st
4  Allison, Mrs. Hudson J C (Bessi    no  female    25.00         1st
      Unnamed: 0 survived    sex    age passengerClass
1304    Zabour, Miss. Hileni    no  female    14.5         3rd
1305    Zabour, Miss. Thamine    no  female     NaN         3rd
1306  Zakarian, Mr. Mapriededer    no   male    26.5         3rd
1307    Zakarian, Mr. Ortin    no   male    27.0         3rd
1308    Zimmerman, Mr. Leo    no   male    29.0         3rd
```

Viewing Some of the Rows in the Titanic Dataset (cont.)

- Note that pandas adjusts each column's width, based on the widest value in the column or based on the column name, whichever is wider.
- Also, note the value in the age column of row 1305 is NaN (not a number), indicating a missing value in the dataset.

Customizing the Column Names

- The first column in this dataset has a strange name ('Unnamed: 0').
- We can clean that up by setting the column names.
- Let's change 'Unnamed: 0' to 'name' and let's shorten 'passengerClass' to 'class':

```
import pandas as pd

titanic = pd.read_csv('https://vincentarelbundock.github.io/' +
                      'Rdatasets/csv/carData/TitanicSurvival.csv')
pd.set_option('precision', 2)
titanic.columns = ['name', 'survived', 'sex', 'age', 'class']
print(titanic.head())
```

Customizing the Column Names (cont.)

- Output

	name	survived	sex	age	class
0	Allen, Miss. Elisabeth Walton	yes	female	29.00	1st
1	Allison, Master. Hudson Trevor	yes	male	0.92	1st
2	Allison, Miss. Helen Loraine	no	female	2.00	1st
3	Allison, Mr. Hudson Joshua Crei	no	male	30.00	1st
4	Allison, Mrs. Hudson J C (Bessi	no	female	25.00	1st

Simple Data Analysis with the Titanic Disaster Dataset

- Now, you can use pandas to perform some simple analysis. For example, let's look at some descriptive statistics.
- When you call `describe` on a DataFrame containing both numeric and non-numeric columns, `describe` calculates these statistics *only for the numeric columns*—in this case, just the age column:

```
import pandas as pd

titanic = pd.read_csv('https://vincentarelbundock.github.io/' +
                      'Rdatasets/csv/carData/TitanicSurvival.csv')
pd.set_option('precision', 2)
titanic.columns = ['name', 'survived', 'sex', 'age', 'class']
print(titanic.describe())
```

Simple Data Analysis with the Titanic Disaster Dataset (cont.)

- Output

	age
count	1046.00
mean	29.88
std	14.41
min	0.17
25%	21.00
50%	28.00
75%	39.00
max	80.00

Simple Data Analysis with the Titanic Disaster Dataset (cont.)

- Note the discrepancy in the count (1046) vs. the dataset's number of rows (1309— the last row's index was 1308 when we called tail).
- Only 1046 (the count above) of the records contained an age value.
- The rest were *missing* and marked as NaN, as in row 1305.
- When performing calculations, Pandas *ignores missing data (NaN) by default*.
- For the 1046 people with valid ages, the average (mean) age was 29.88 years old.

Simple Data Analysis with the Titanic Disaster Dataset (cont.)

- The youngest passenger (min) was just over two months old ($0.17 * 12$ is 2.04), and the oldest (max) was 80.
- The median age was 28 (indicated by the 50% quartile).
- The 25% quartile is the median age in the first half of the passengers (sorted by age), and the 75% quartile is the median of the second half of passengers.

Simple Data Analysis with the Titanic Disaster Dataset (cont.)

- Let's say you want to determine some statistics about people who survived.
- We can compare the survived column to 'yes' to get a new Series containing True/False values, then use describe to summarize the results:

```
import pandas as pd

titanic = pd.read_csv('https://vincentarelbundock.github.io/' +
                      'Rdatasets/csv/carData/TitanicSurvival.csv')
pd.set_option('precision', 2)
titanic.columns = ['name', 'survived', 'sex', 'age', 'class']
print((titanic.survived == 'yes').describe())
```

Simple Data Analysis with the Titanic Disaster Dataset (cont.)

- Output

```
count      1309
unique         2
top        False
freq         809
Name: survived, dtype: object
```

- For non-numeric data, describe displays different descriptive statistics:
 - count is the total number of items in the result.
 - unique is the number of unique values (2) in the result—True (survived) and False (died).
 - top is the most frequently occurring value in the result.
 - freq is the number of occurrences of the top value.

Matplotlib

- Visualization is a nice way to get to know your data.
- Pandas has many built-in visualization capabilities that are implemented with Matplotlib. To use them, first install Matplotlib:

```
conda install matplotlib
```

- Importing matplotlib:

```
import matplotlib.pyplot as plt
```

Passenger Age Histogram

- A histogram visualizes the distribution of numerical data over a range of values. A DataFrame's **hist** method automatically analyses each numerical column's data and produces a corresponding histogram.
- To view histograms of each numerical data column, call hist on your DataFrame and call show from matplotlib pyplot:

```
import pandas as pd
import matplotlib.pyplot as plt

titanic = pd.read_csv('https://vincentarelbundock.github.io/' +
                      'Rdatasets/csv/carData/TitanicSurvival.csv')
pd.set_option('precision', 2)
titanic.columns = ['name', 'survived', 'sex', 'age', 'class']
titanic.hist()
plt.show()
```


Passenger Age Histogram (cont.)

- The Titanic dataset contains only one numerical data column, so the diagram below shows one histogram for the age distribution.
- For datasets with multiple numerical columns (as we'll see in the exercises), hist creates a separate histogram for each numerical column.

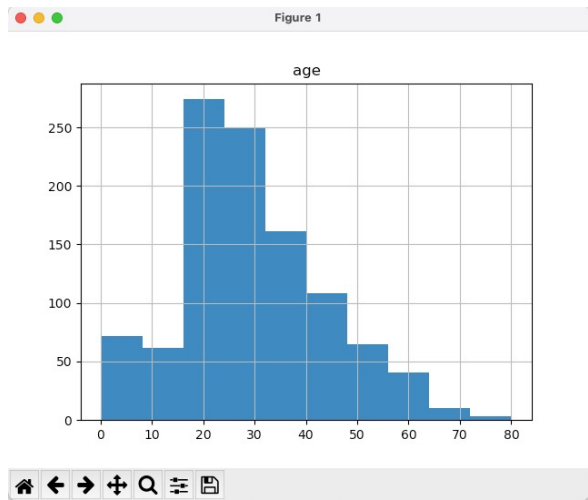


Chart Visualisation

- There are more charts that can be plotted using Pandas, you may refer to the documentation.
- https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html

Thanks

Do you have any question?

jailani.rahman@pb.edu.bn
(+673) 223 4466 ext 241
Unit 6.01, Ong Sum Ping Campus,
Politeknik Brunei.

CREDITS: This presentation template was created by **Slidesgo**,
including icons by **Flaticon**, infographics & images by **Freepik**