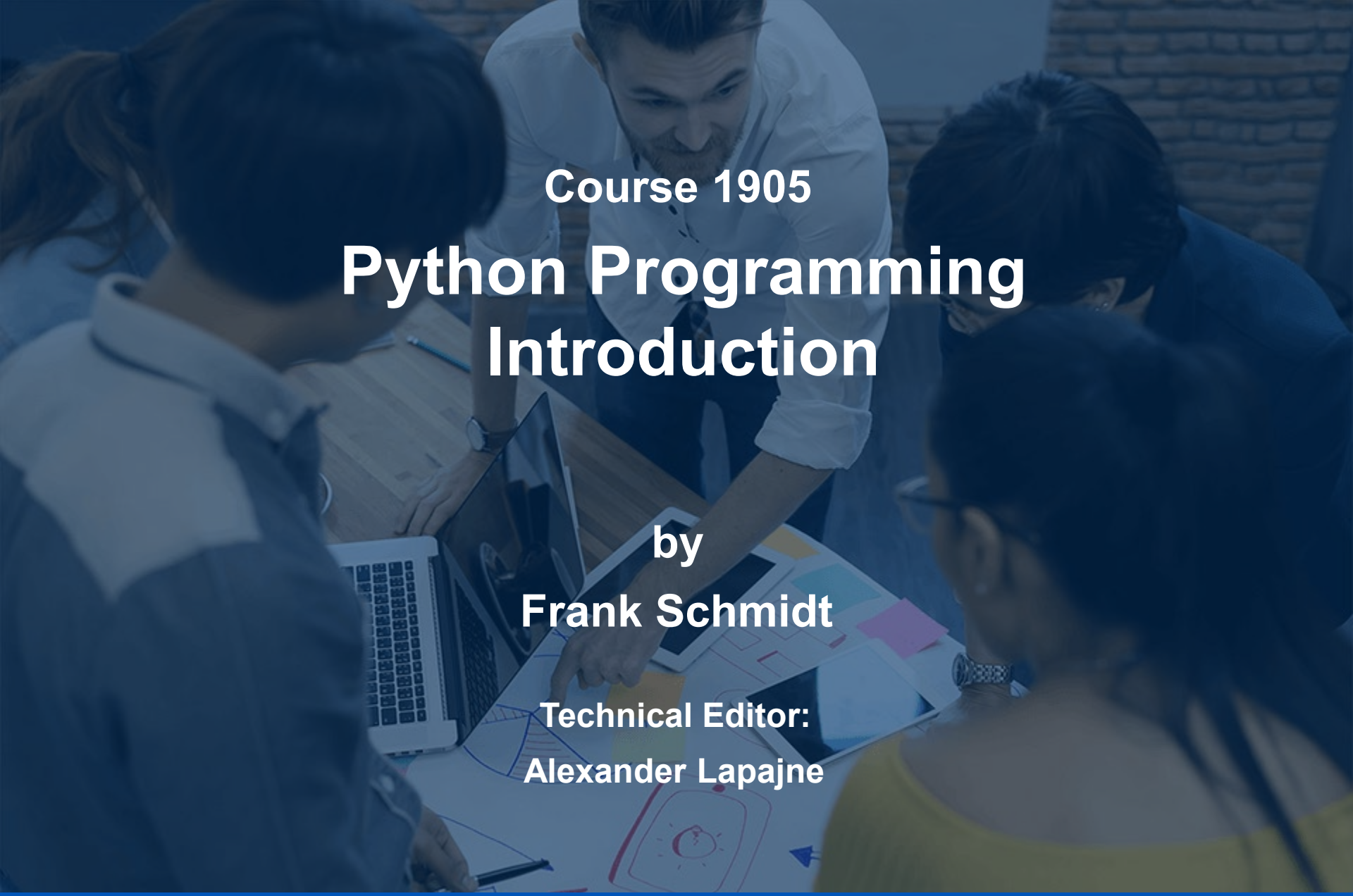


Course 1905

Python Programming Introduction

by

Frank Schmidt

Technical Editor:

Alexander Lapajne

LEARNING TREE INTERNATIONAL

Copyright

Introduction and Overview

LEARNING TREE INTERNATIONAL

Course Objectives

- Create, edit, and execute Python programs in PyCharm
- Use Python simple data types and collections of these types
- Control execution flow: conditional testing, loops, and exception handling
- Encapsulate code into reusable units with functions and modules
- Employ classes, inheritance, and polymorphism for an object-oriented approach
- Read and write data from files
- Query databases using SQL statements within a Python program

SQL = structured query language

Course Contents

Chapter 1

Python Overview

LEARNING TREE INTERNATIONAL

Objectives

- Describe the uses and benefits of Python
- Enter statements into the Python console
- Create, edit, and execute Python programs
- Identify sources of documentation

Contents

Python Background

- Documentation Resources

A Definition of Python

- **An object-oriented, open-source programming language**
  - Inheritance, encapsulation, and polymorphism supported
  - Freedom to use Python and its applications for no charge
- **A general-purpose language used for a wide variety of application types**
  - Text and numeric processing
  - Operating system utilities and networking through the standard library
  - GUI and web applications through third-party libraries
- **Python Software Foundation (PSF) controls the copyright and development of the language after version 2.1**
  - An independent nonprofit group
  - Guido van Rossum started creating the language in 1989

Python Philosophy

- **Simple solutions**
  - Code should clearly express an idea or task
- **Readability**
  - White space and indentation to define blocks of code
  - Less coding required as compared to the equivalent .NET, C++, or Java
    - Decreases development and maintenance time requirements
- **Dynamic typing and polymorphism**
  - No data type declarations
  - Operator and method overloading
    - Operators are evaluated at runtime based on the expression type

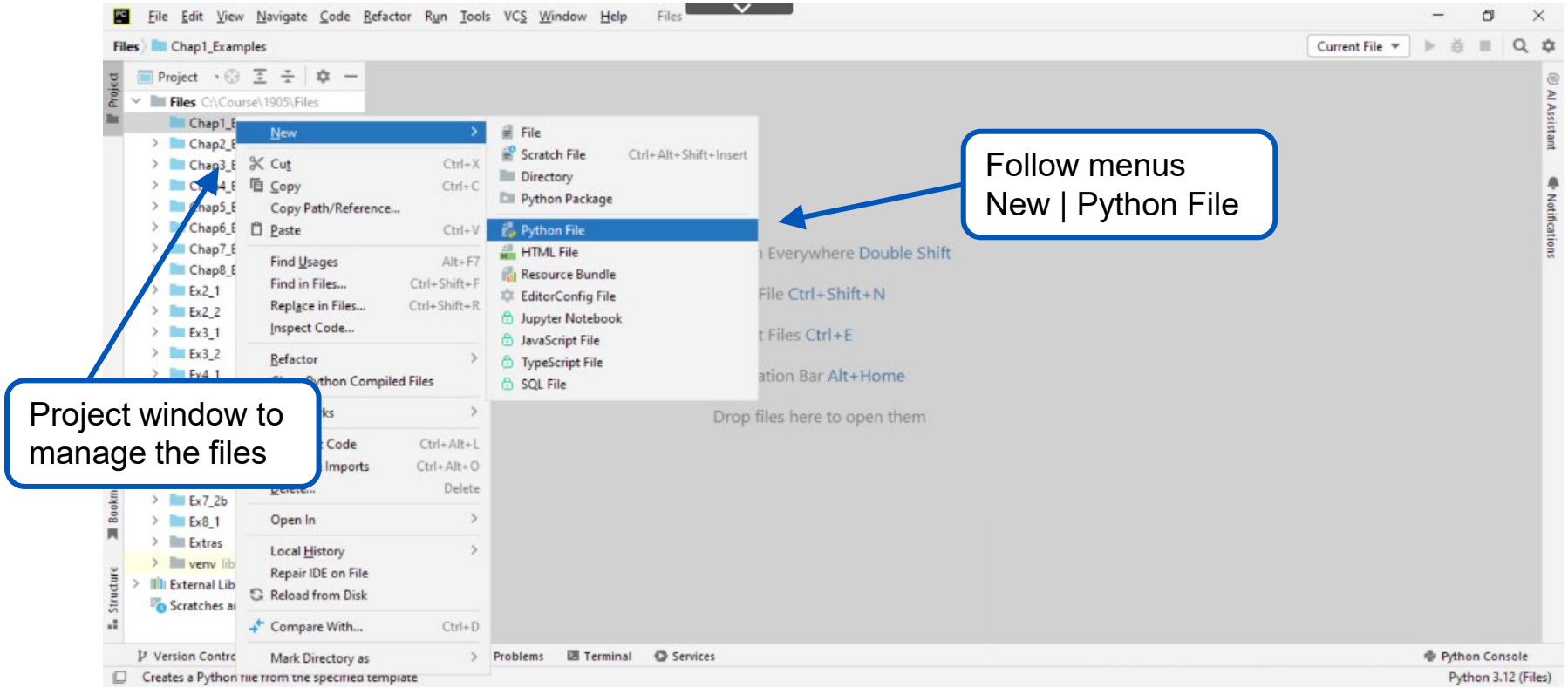
The Python Interpreter

- **Executes source code statements**
- **Accessing the interpreter**
  - From the OS command prompt
    - `python`
    - `python` *`filename`*`.py`
  - From an Integrated Development Environment, IDE
    - Includes the interpreter to execute `.py` files
    - Includes an editor to create and modify `.py` files
    - Includes additional tools:
      - File browser
      - Interactive console
      - Variable browser
      - Debugger

Running Python Programs in PyCharm

Do Now

1. Access PyCharm using the button on the taskbar
2. From the Project window on the left, right-click the `Chap1_Examples` folder and select New | Python File
3. Name the file `first.py`



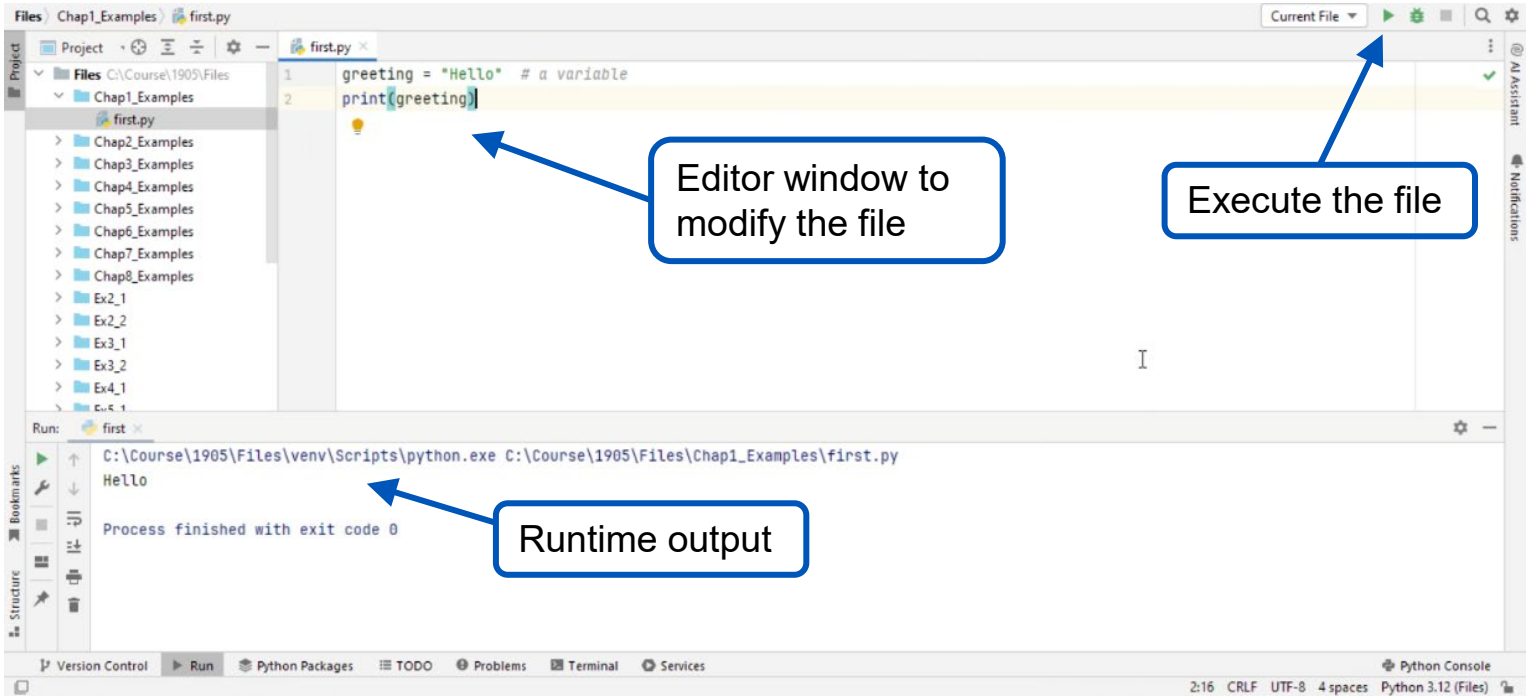
Editing and Running Python Programs in PyCharm

Do Now

4. **Editor window opens, input the following lines into the Editor window:**

```
greeting = "Hello"  # a variable
print(greeting)
```

5. **Click the Green Arrow to run the Current File, `first.py`**
   - Output appears in the Run window at the bottom

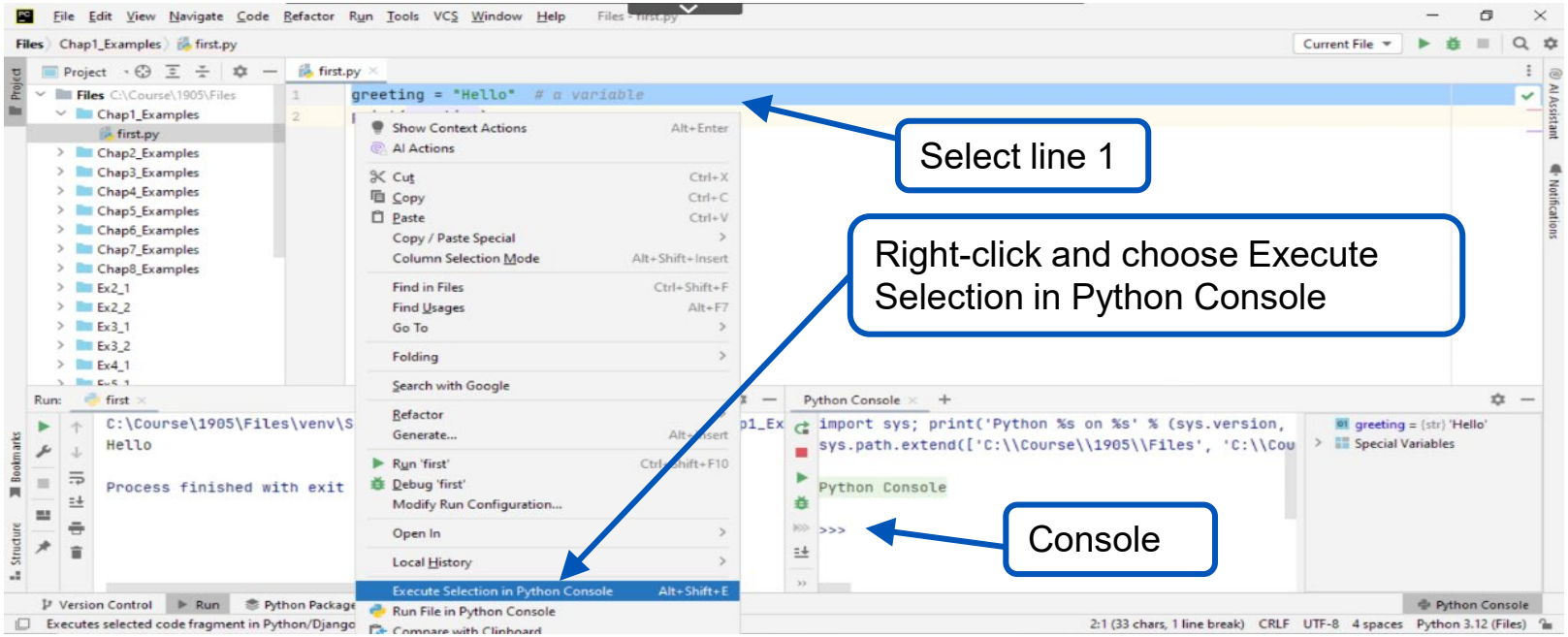


Using Python Console in PyCharm

Do Now

- **Console interactively executes statements**
  - Without changing the editor contents

6. **Use the Left mouse button to highlight and select the first line of source**
7. **Use the Right mouse button to select Execute Selection in Python Console**
8. **Notice the Python Console opens below**



Using Python Console in PyCharm

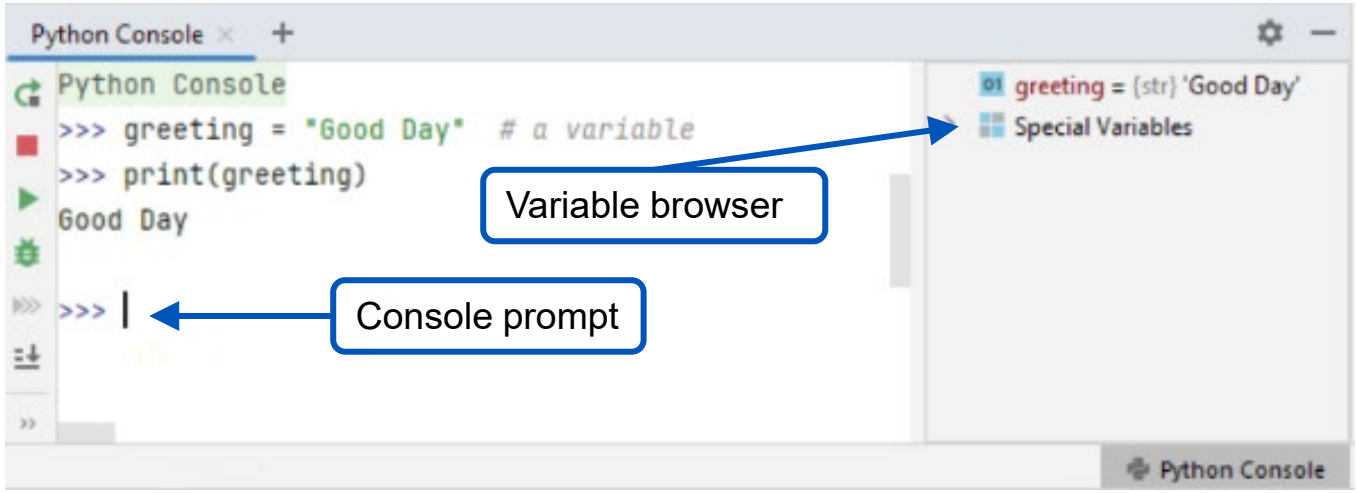
Do Now

9. **Notice the variable browser pane on the right with the variable `greeting`**

10. **Use the Up Arrow key to retrieve the previous `greeting = "Hello"` statement**

11. **Modify the statement to read `greeting = "Good Day"`**

12. **Press <Enter>**
    - The variable has been changed
    - Enter `print(greeting)` at the next console prompt
    - Output is displayed below



PyCharm

- **Use the Editor and Run button to develop and execute entire `.py` files**
  - Hands-On activities
  - Creating an application that may composed of many files
- **Use the Console to:**
  - Execute portions of `.py` files
  - Modify previous statements from within the console
  - Interactively enter additional Python statements
  - Experiment and practice

Contents

- Python Background

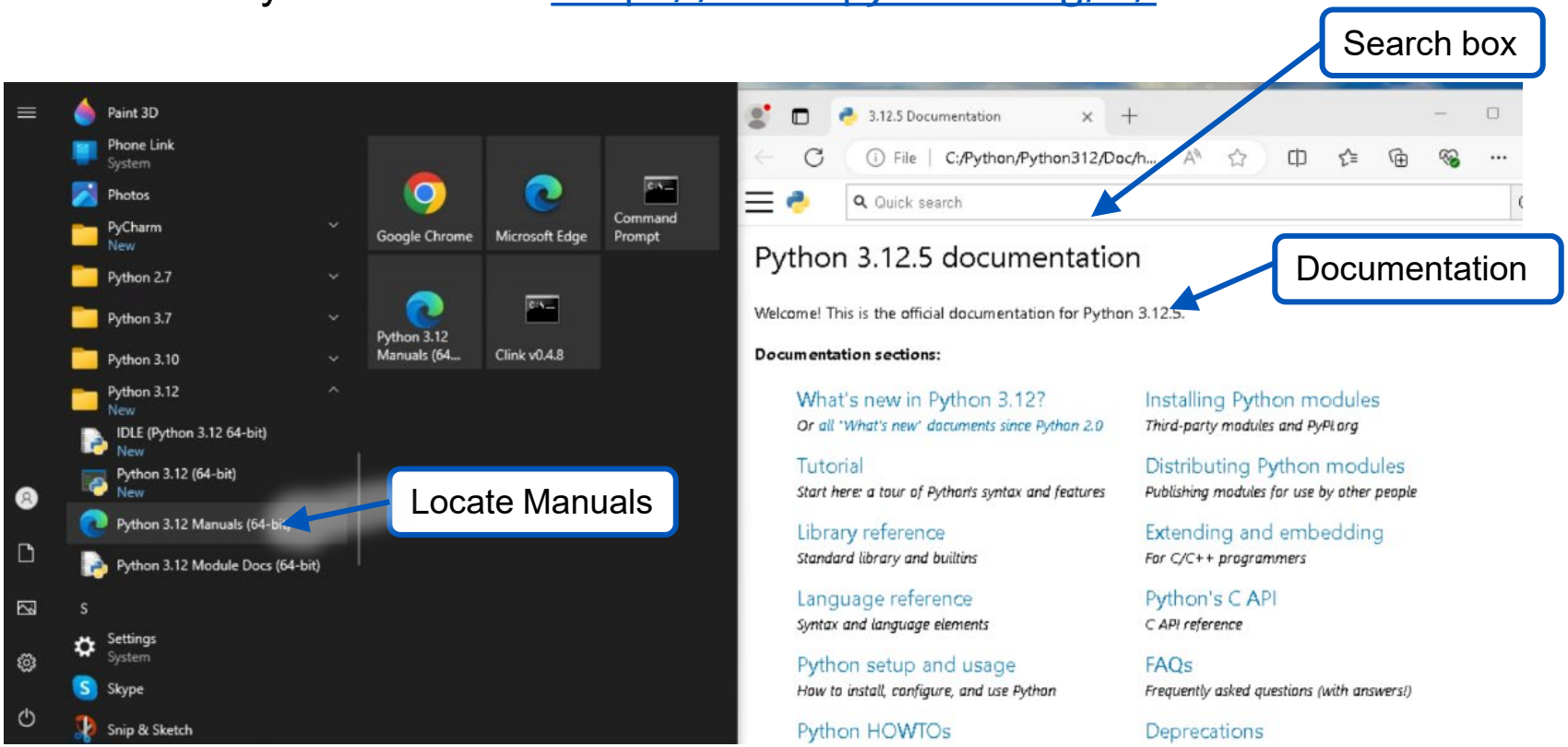
Documentation Resources

Source Code Documentation

- **Comments in source code follow the # character**
  - Remainder of the line is ignored
  - No block format
- **Doc strings**
  - Describe modules, functions, classes
  - Enclosed in a set of triple quotation marks
  - Available as the `__doc__` attribute of an object

Python Documentation

- **Official documentation is locally installed on the virtual machine**
  - Under Python 3.12 | Python 3.12 Manuals
  - Viewed with the default web browser
  - Externally available via http://docs.python.org/3/

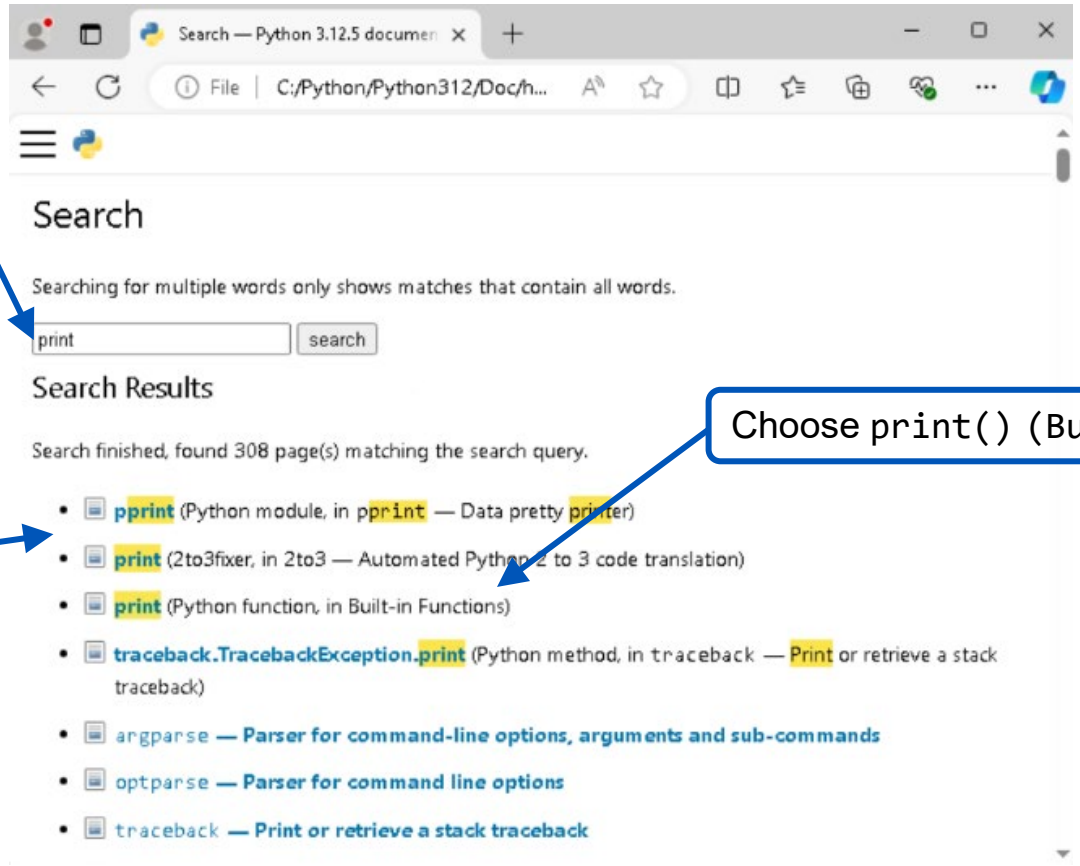


HTML = HyperText Markup Language

Using Documentation

Do Now

1. **Access Python documentation from the Windows Start menu**
2. **Enter `print` into the search box**
3. **Select `print()` (`built-in function`)**



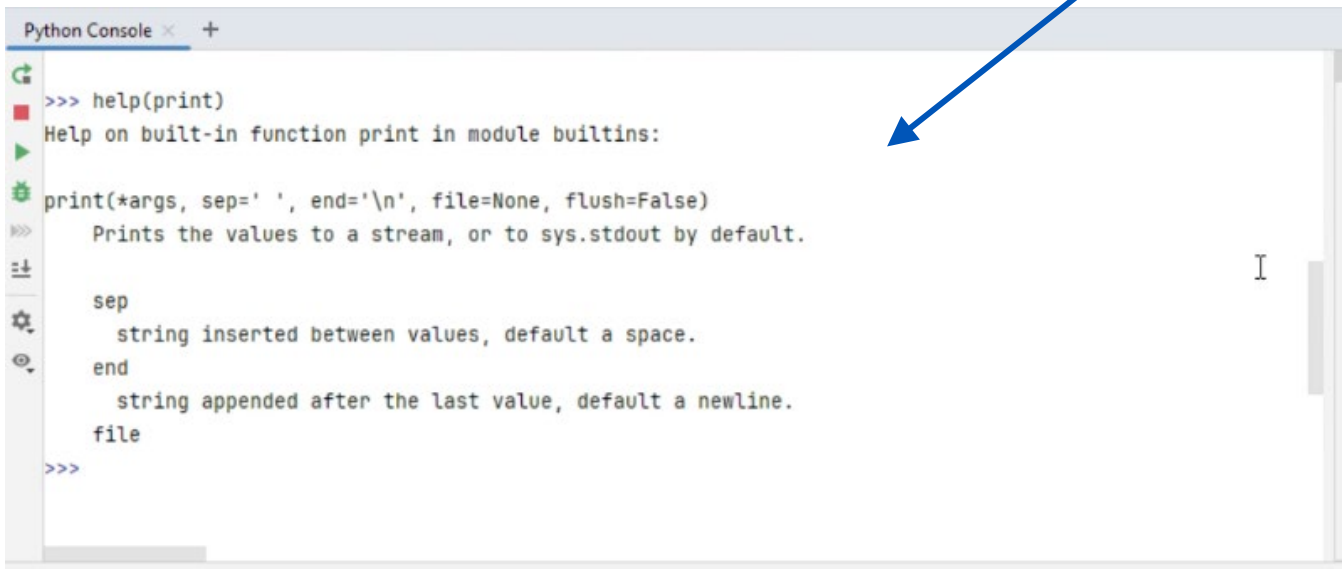
Search box term

Manual entries that match search

Choose `print()` (`Built-in Function`)

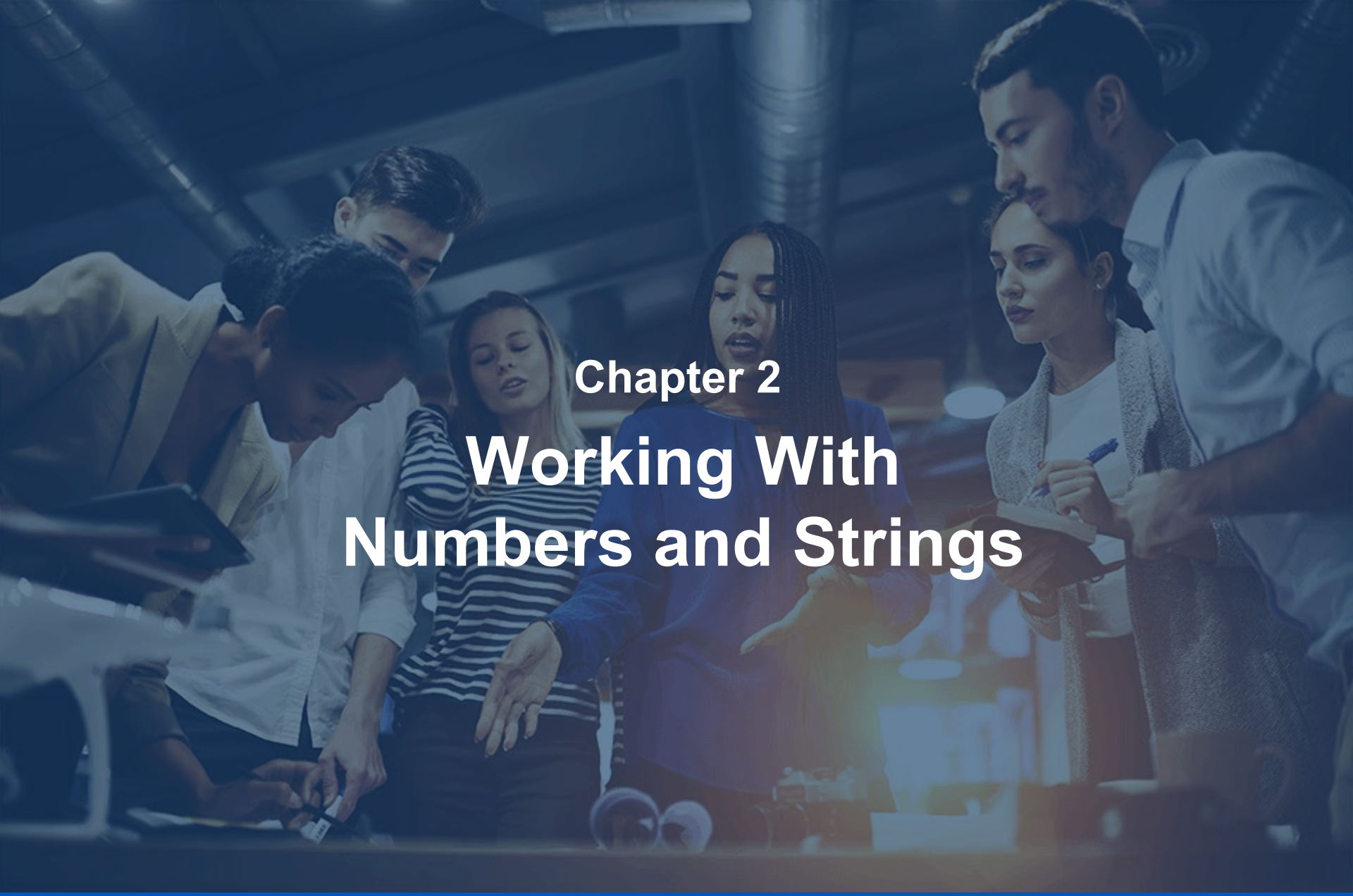
help()

- **Built-in Python function provides a console interface to documentation about modules, functions, keywords, etc.**
  - `help()`
    - To start the help system
    - `quit` to exit help
  - `help(`*`object`*`)`
    - To access the documentation for that object
    - `help(print)`



Objectives

- Describe the uses and benefits of Python
- Enter statements into the Python console
- Create, edit, and execute Python programs
- Identify sources of documentation



Chapter 2

Working With Numbers and Strings

LEARNING TREE INTERNATIONAL

Objectives

- **Write simple Python programs**
  - Create variables
  - Manipulate numeric values
  - Manipulate string values
  - Make decisions

Contents

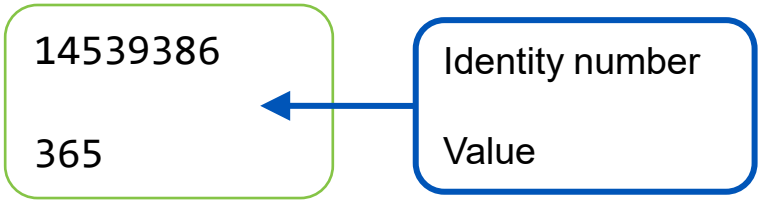
Objects and Variables

- Numeric Types and Operations
- String Types and Operations
- Conditionals

Python Objects

- **An object is an instance of a data value stored in a memory location**
  - Memory is allocated when the object is created
    - Built-in function `id()` returns an integer representing its identity
  - Memory is reclaimed when the object is no longer referenced
    - *Garbage collection*
  - `1`, `2.5`, and `'Welcome'` are all objects

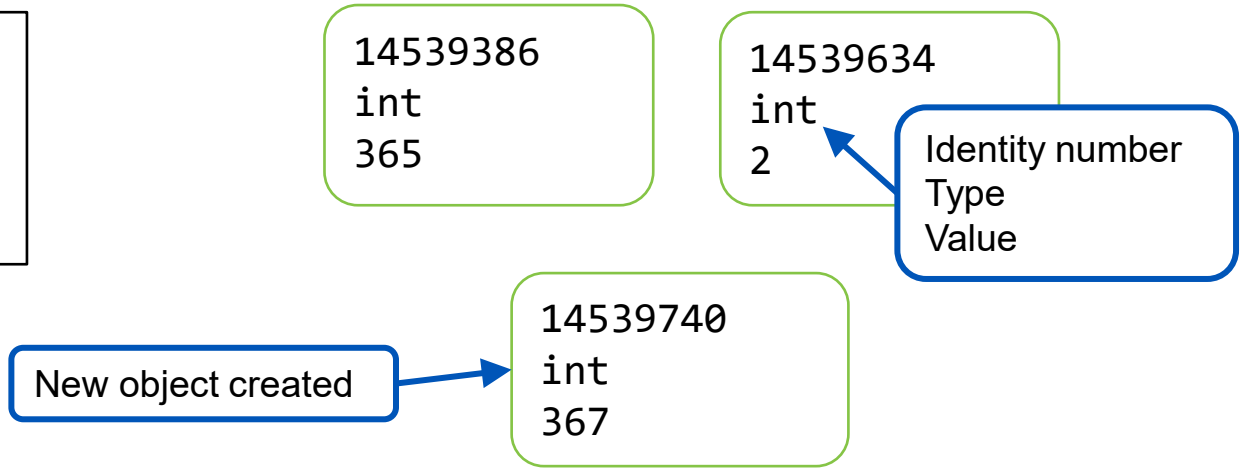
```
>>> 365
365
>>> id(365)
14539386
```



Python Object Type

- **An object has a *type***
  - Built-in function `type()` shows type
  - `1` is an `int` type—digits
  - `2.5` is a `float` type—digits with a decimal
  - `'Welcome'` is a `str` type—enclosed in quote marks
- **An object's type constrains the operations on that object**
  - Arithmetic on integer or floating-point types
  - Concatenation on a string type

```
>>> type(365)
<class 'int'>
>>> 365 + 2
367
```

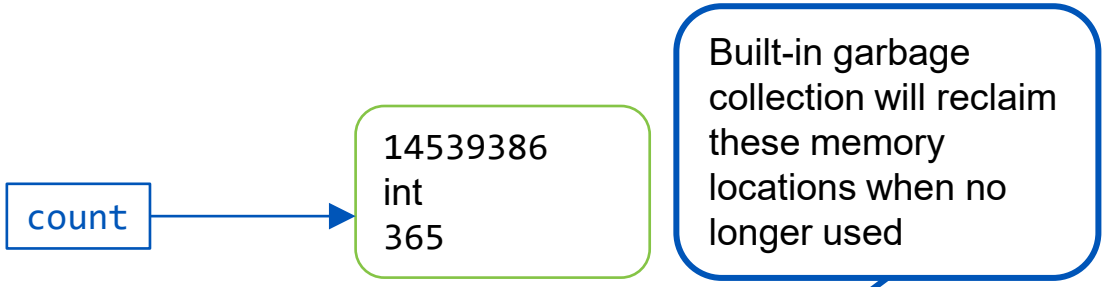


Variables

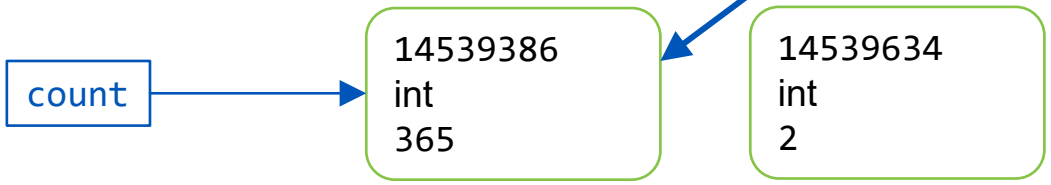
- **A variable is a named reference to an object**
  - Operations on variables use the object referenced
  - Operations are constrained by the type of the object
  - Variable instance is created when an object is assigned
    - An *identifier*
- **A variable is modified by assigning a different object**
- **May reference any type of object**

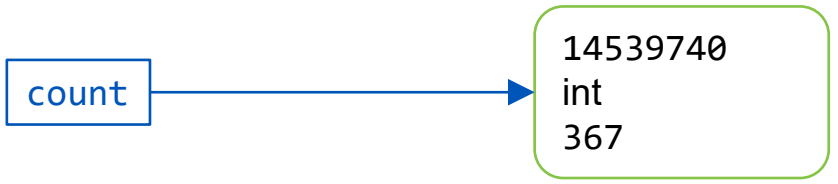
Variables Illustrated

```
>>> count = 365
>>> count
365
```




```
>>> count = count + 2
>>> count
367
```






```
>>> count = "Dracula"
>>> count
'Dracula'
```

Naming Rules

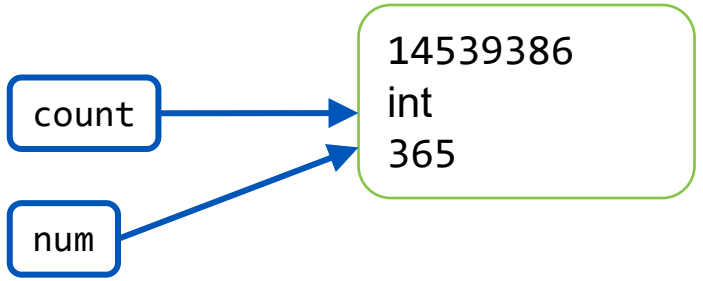
- **Start with letter or underscore**
  - Followed by any number of letters, digits, or underscores
    - Case sensitive
    - Leading and trailing underscores are special
- **May not be a keyword**
  - Should not be a built-in name
- **PEP 8, the style guide for Python, recommends lowercase**
  - `firstname`
  - `first_name`

Shared Reference

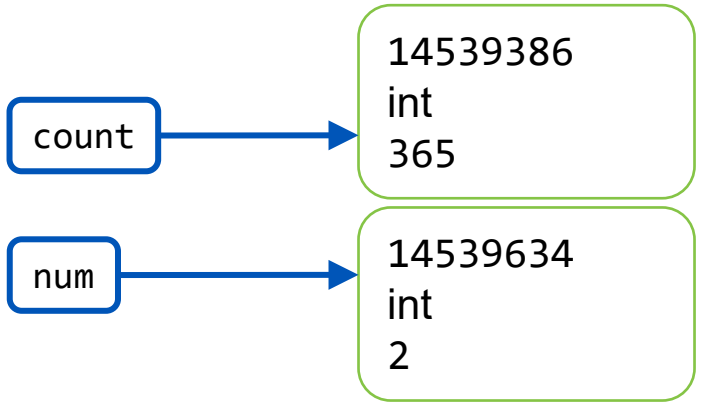
- **Multiple variables reference the same object**
  - Created by assigning one variable to another
    - Or the same object to both
  - Confirmed through object identity test operator `is`

```
>>> count = 365
>>> count
365
>>> num = count
>>> num
365
>>> num is count
True
```

Boolean result




```
>>> num = 2
>>> num
2
>>> num is count
False
```



Contents

- Objects and Variables

Numeric Types and Operations

- String Types and Operations
- Conditionals

Python Built-In Types

- **Immutable types**
  - String literals
  - Arithmetic literals
    - Integer, floating point
- **Collection types**
  - Lists, dictionaries, and sets are mutable
  - Tuples are immutable

Numeric Objects

- **Numbers are a core Python type**
- **Integers can be represented exactly in memory and have no fractional part**
  - Examples:

    ```
    4        -3        12
    ```

  - Decimal, base 10, is default
    - Octal with leading 0o
    - Hexadecimal with leading 0x
    - Binary representation with leading 0b
      - Example: `0o14`, `0xC`, and `0b1100` are equivalent to decimal `12`
- **Floating point objects have an integer portion and a fractional portion**
  - Examples:

    ```
    4.0     123.56     0.000001       1.5e10       6.9E-6
    ```

  - Floating point objects are represented as approximations in memory
- **Complex number objects are stored as two floating point values**
  - For the real and imaginary parts

Numeric Operators

`( )`	Raise precedence level
`a ** b`	Exponentiation

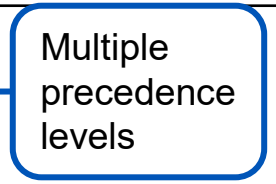
`~a`
`-a`
`+a`
	Bitwise NOT
	Negation
	Unchanged

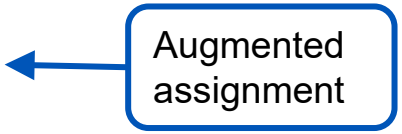
`a * b`
`a / b`
`a // b`
`a % b`
	Multiplication
	True division
	Floor division
	Modulus

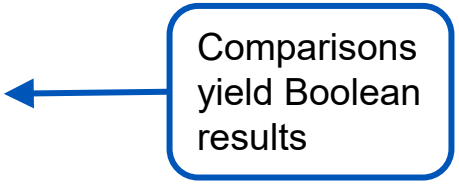
`a + b` `a - b`	Addition Subtraction
`a << b, a >> b`	Bit shift
`a & b`	Bitwise AND
`a ^ b`	Bitwise exclusive OR
`a | b`	Bitwise OR
`a < b, a <= b, a > b, a >= b` `a != b, a == b`	Relational Equality
`=, +=, -=, *=, /=, %=`	Assignment

Numeric Operators Example

```
>>> count = 3
>>> count = count + 3 * 5
>>> count
18
>>> count += 1
>>> count
19
>>> num = 2
>>> num < count
True
>>> count == 3
False
```

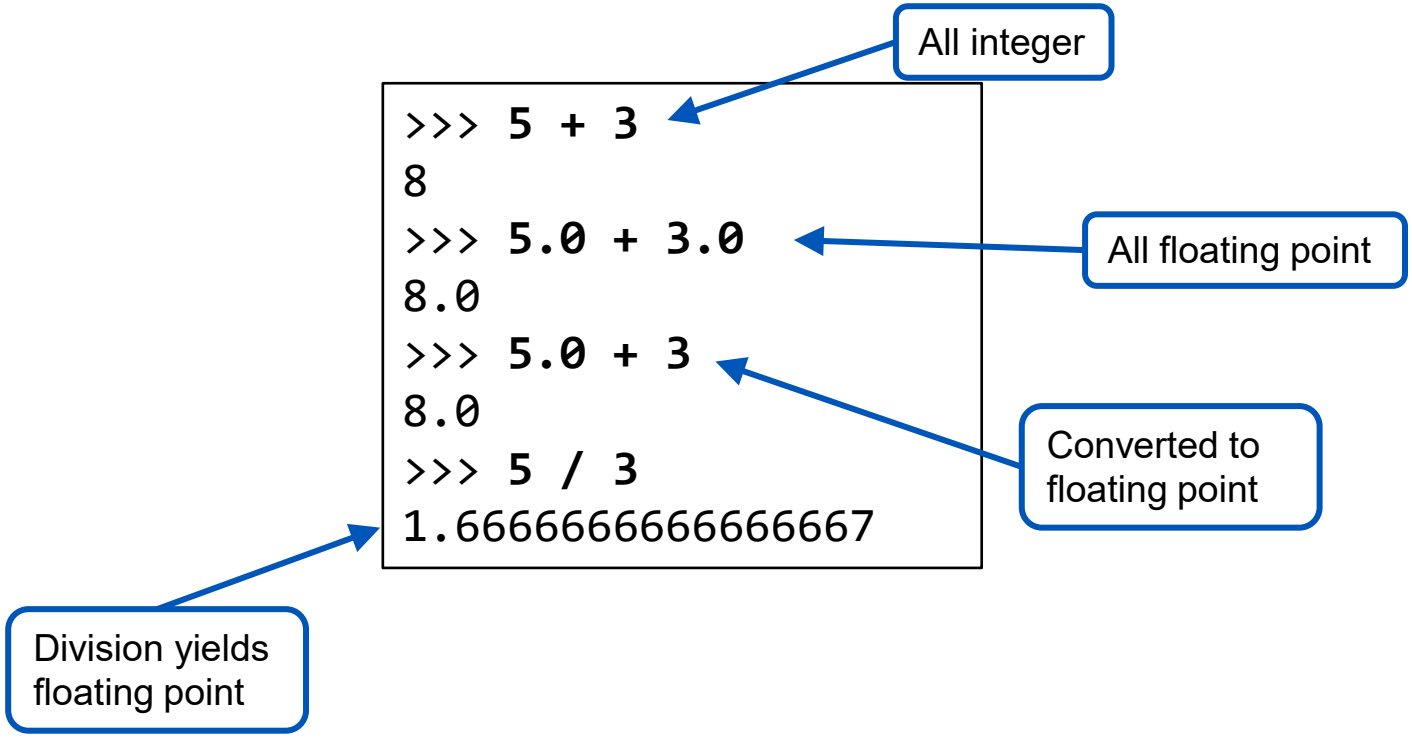






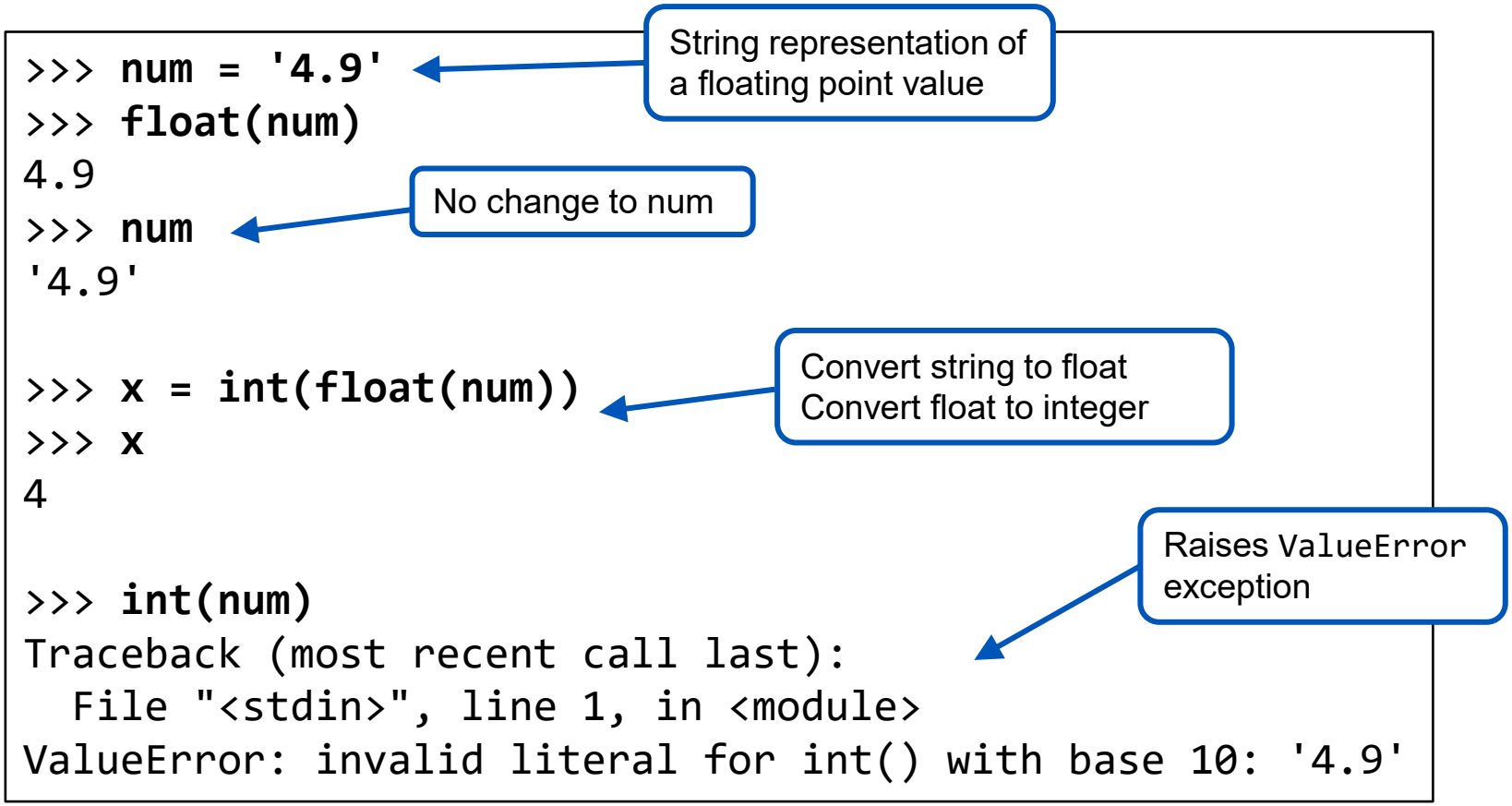
Numeric Operation Type

- **Operations on objects of the same type yield results of the same type**
  - Except division, which yields a floating-point result in Python 3.x
- **Results of operations on objects of mixed types are converted to the bigger type**



Arithmetic Typing Functions

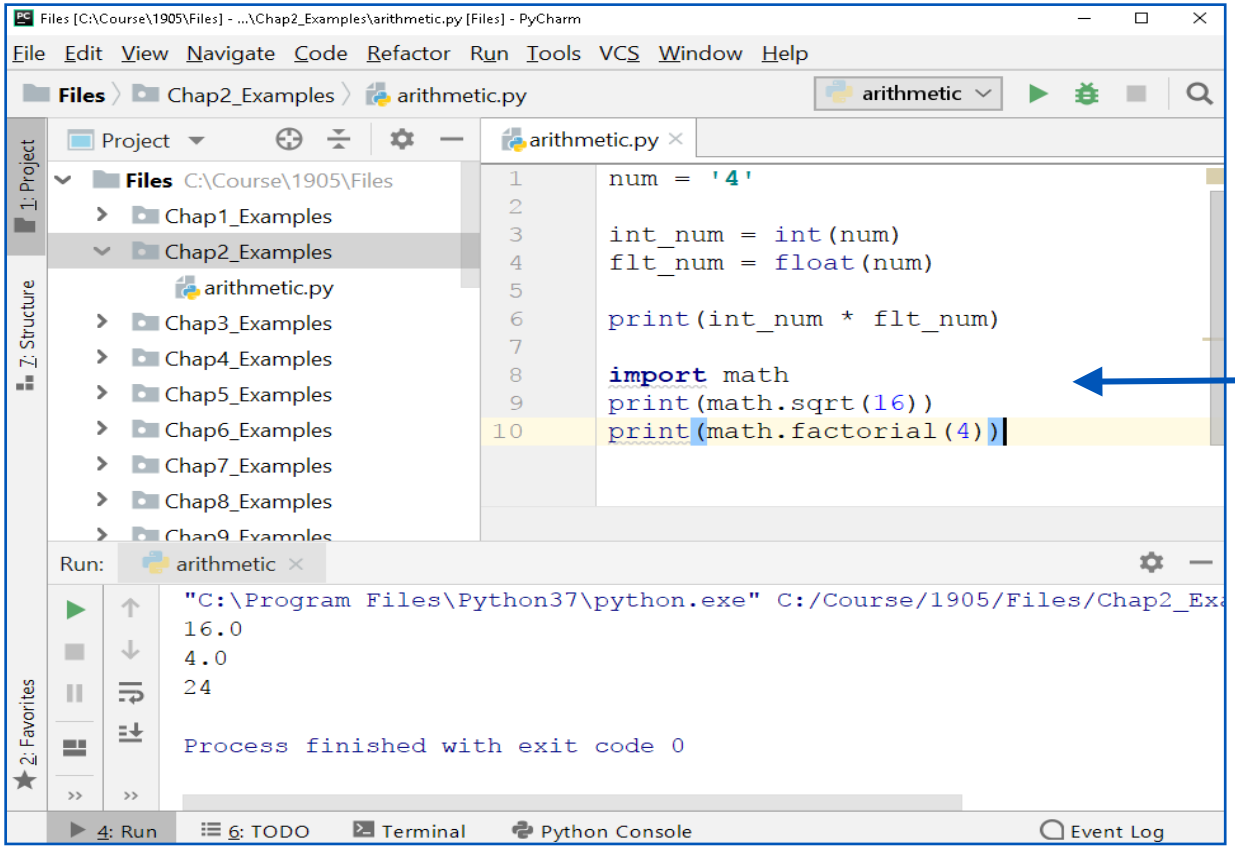
- **The `float()` and `int()` functions return the argument in the specified type**
  - Argument may be the string representation of a numeric value
  - Argument may be an arithmetic expression



The math Module

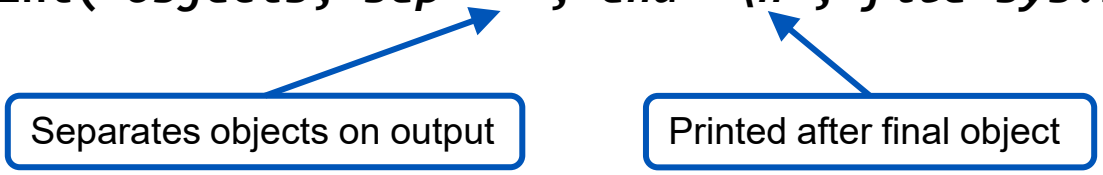
Reference

- **Provides many additional arithmetic capabilities**
- **Is part of Python's standard library**
  - `import` to make the functions available
    - References to objects within the module require a qualified name



print Function

- `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`



- **Print *objects* to the text stream *file*, separated by *sep* and followed by *end***

```
>>> name = 'Ian'
>>> age = 34
>>> print('Hello', name, 'your age is', age)
Hello Ian your age is 34
>>> print(name, age, sep='-->')
Ian-->34
>>> print(name, age, sep='')
Ian34
```

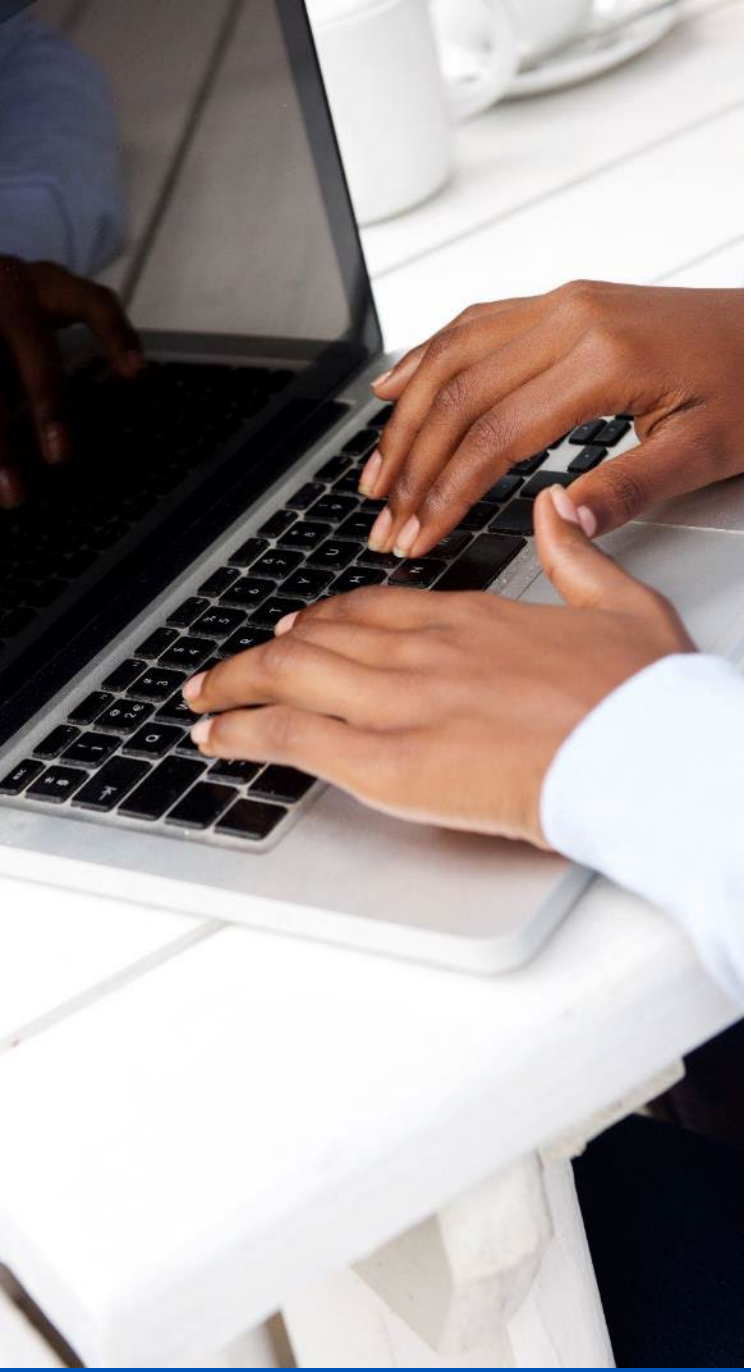
Default *sep* and *end*

Output object separator

Empty string separator between objects

AdaptaLearn™ Enabled

- **Interactive exercise manual that offers an enhanced learning experience**
  - Folded steps reveal hints and answers
  - Code can be copied from the manual when necessary
  - After class, the manual can be accessed remotely for continued reference and practice
- **Printed and downloaded copies show all detail levels (hints and answers are unfolded)**
- **Accessing AdaptaLearn:**
  1. Launch AdaptaLearn by double-clicking its icon on the desktop
     - Move the AdaptaLearn window to the side of your screen or shrink it to leave room for a work area for your development tools
  2. Select an exercise from the exercise menu
     - Zoom in and out of the AdaptaLearn window
     - Toggle between the AdaptaLearn window and your other windows
- **Your instructor can answer any questions you have about AdaptaLearn functions**



Hands-On Exercise 2.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 2.1: Arithmetic and Numeric Types

Contents

- Objects and Variables
- Numeric Types and Operations

String Types and Operations

- Conditionals

String Objects

- **String values are defined between a pair of quotation marks**
  - Single and double quotes are equivalent
  - Triple quotes of either type are allowed

```
>>> name = 'Guido'
>>> question = "Don't you love Python?"
>>> question = '''Guido asked "Don't you love Python?"'''
```

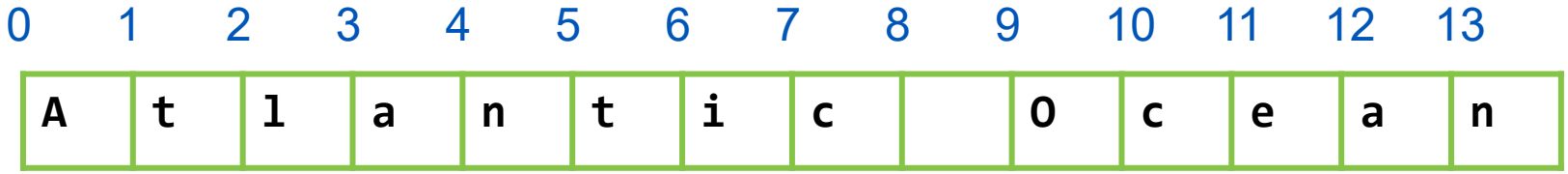
- **Strings are the core Python type `str`**
  - Sequence type
    - Series of single characters ordered left to right by position
    - Individual characters can be referenced
  - Immutable type
  - Object cannot be changed

- **String values represent a series of Unicode code points**
  - Characters are one to four bytes in length
  - ASCII is a subset of Unicode

Substrings

- **Individual characters are referenced by offset**
- **A slice is a portion of a sequence**
  - Described by its offset
  - Slice boundary is a range specified in [*start*:*end*]
  - Slice of a string is another string

Non-inclusive of the character at the *end* offset

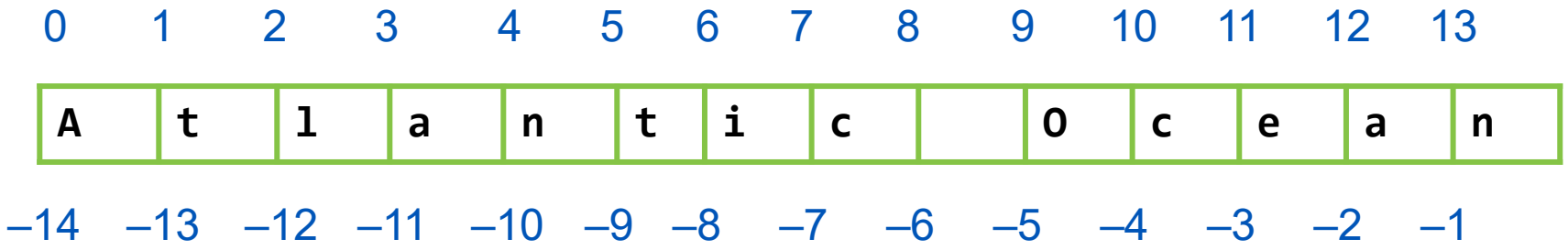



```
>>> sea = 'Atlantic Ocean'
>>> sea[0]
'A'
>>> sea[0:8]
'Atlantic'
>>> sea[:8]
'Atlantic'
>>> sea[9:]
'Ocean'
```

Unbounded slices extend to an end

String Slicing

- **An offset may be described from either end of the string**
  - Use a negative offset



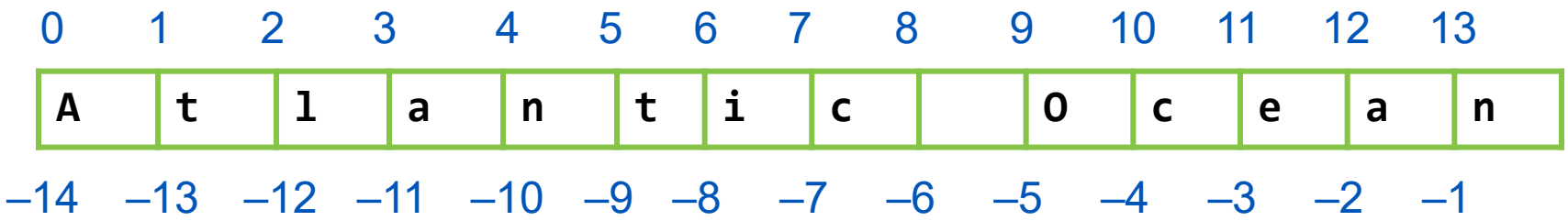

```
>>> sea[-1]
'n'
>>> sea[-5:]
'Ocean'
>>> sea[-1:-5]
''
```

Always the last reference from a sequence

Undefined slice yields an empty string

String Slicing

- **A step or stride may access nonsequential values**
  - Use [*start*:*end*:*step*]




```
>>> sea[1:12:2]
'tatcOe'
>>> sea[-1:-6:-1]
'naecO'
>>> sea[:]
'Atlantic Ocean'
>>> sea[::-1]
'naecO citnaltA'
>>> sea[9] = 'o'
Traceback (most recent call ...
TypeError: 'str' object does not
support item assignment
```

Every second element from 1 to 12

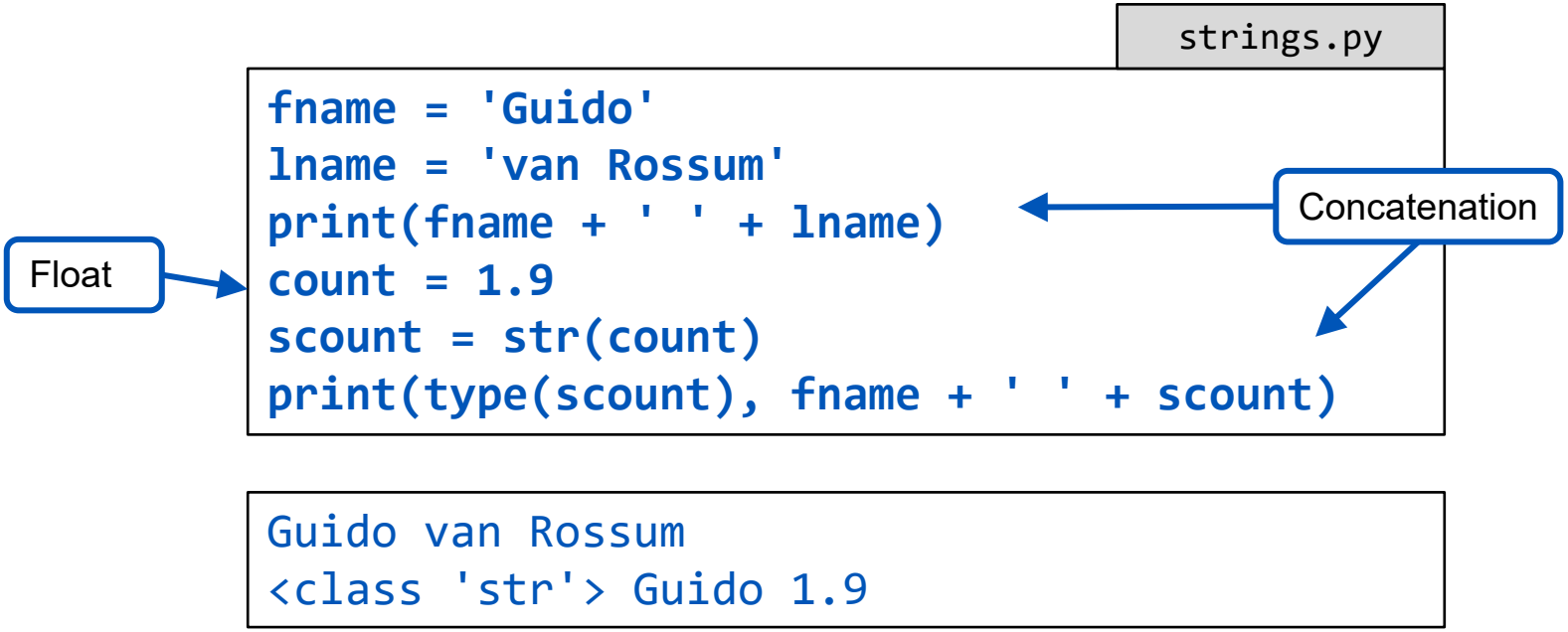
Entire sequence

String reversal

String is immutable

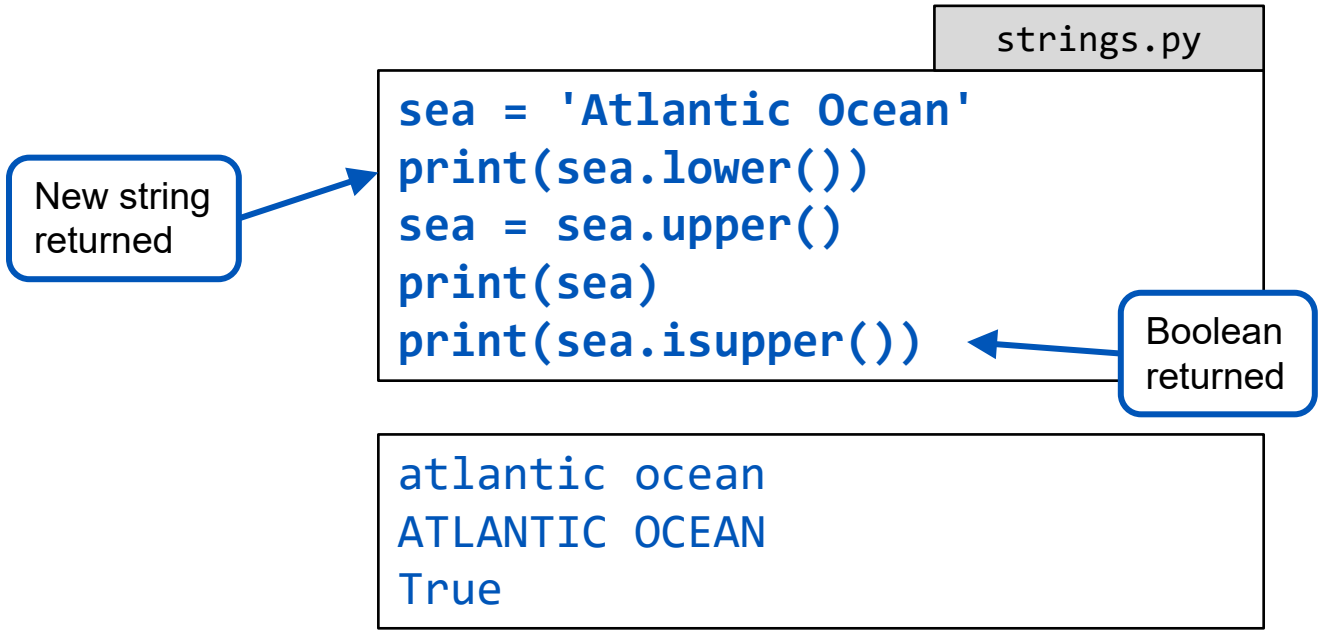
String Concatenation

- **String literals separated by white space are concatenated into a single string**
- **Operators create and return new string objects**
  - + concatenation
  - * repetition
- **The `str()` constructor returns its argument as a string object**



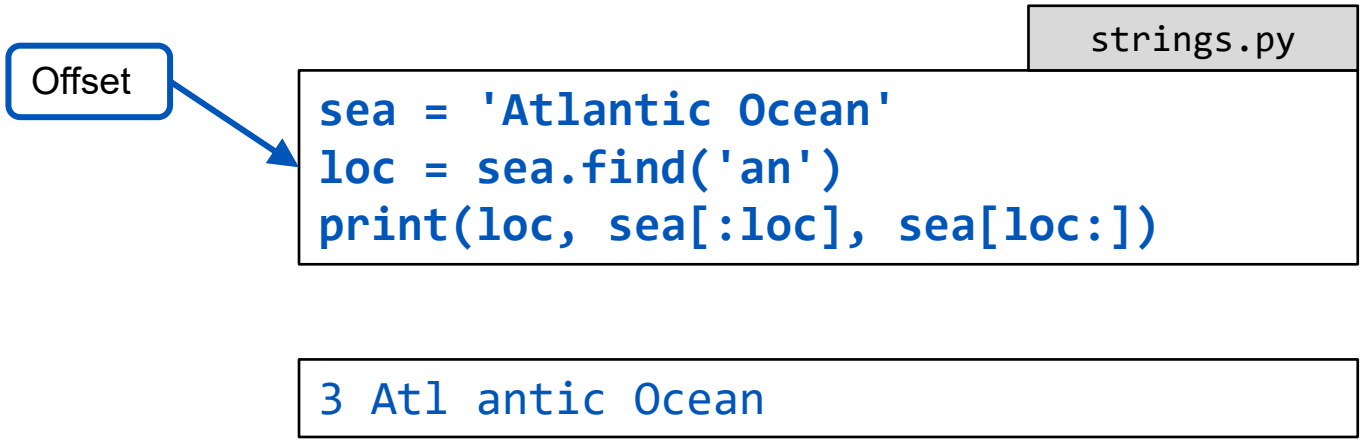
String Methods

- **Functions that operate on string type objects**
  - Syntax: *`string.method()`*
- ***`string`*`.upper()` and *`string`*`.lower()` return a new string**
- ***`string`*`.isupper()` and *`string`*`.islower()` return a Boolean**



String Methods

- ***string*`.find()` returns the offset of the search string**
  - Or `-1` if the string is not found



String Methods

- ***string*.split(*sep*) returns a *list* of strings**
  - Delimited by *sep* string
    - Defaults to any amount of white space
- ***string*.join() returns a delimited string from a sequence**
  - *string* is the delimiter

strings.py

```
sea = 'Atlantic Ocean'
words = sea.split()
print(words)
csv_words = ','.join(words)
print(csv_words)
```

List returned

Delimiter in the returned string

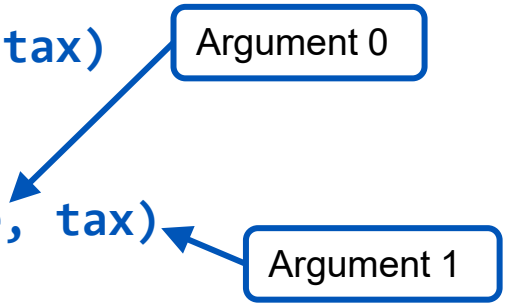
```
['Atlantic', 'Ocean']
Atlantic,Ocean
```

String Formatting

- ***string*.format(*args*) returns a new string after formatting *args***
- ***string* contains text and a series of ordered {} replacement fields**
  - Mapped to *args* by position
- **{*n*} replacement fields identify particular numbered arguments**

strings.py

```
price = 350
tax = 0.07
output = 'price = {}, tax = {}'.format(price, tax)
print(output)

output = 'tax = {1}, price = {0}'.format(price, tax)
print(output)
```




```
price = 350, tax = 0.07
tax = 0.07, price = 350
```

String Formatting

- **{*n*:*spec*} may specify**
  - Width
  - Formatting type code

d	Integer
f	Floating point with precision

strings.py

```
output = 'cost = ${0:7.2f}'.format(price + price * tax)
print(output)
```

Placeholder for argument 0

Format as 7-character floating point with 2 places of precision

```
cost = $ 374.50
```

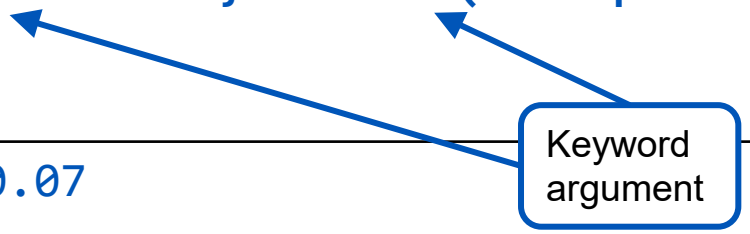
String Formatting

- **{*name*} replacement fields identify particular keyword arguments**
  - Format is the same

strings.py

```
price = 350
tax = 0.07
output = 'price = {p}, tax = {t}'.format(p=price, t=tax)
print(output)

output = 'cost = ${cost:7.2f}'.format(cost=price + price * tax)
print(output)
```

```
price = 350, tax = 0.07
cost = $ 374.50
```

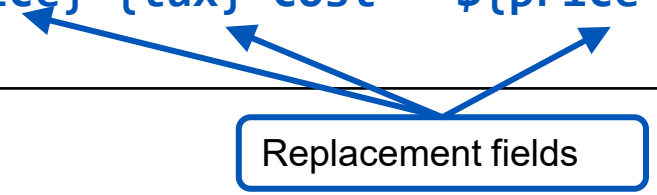


Formatted String Literals

- **Create strings with replacement fields with { }**
  - Format is the same
- **Identified with leading `f` or `F` before quotation marks**
- **Available in Python 3.6 or later**

strings.py

```
price = 350
tax = 0.07
output = f'${price} {tax} cost = ${price + price * tax:7.2f}'
print(output)
```



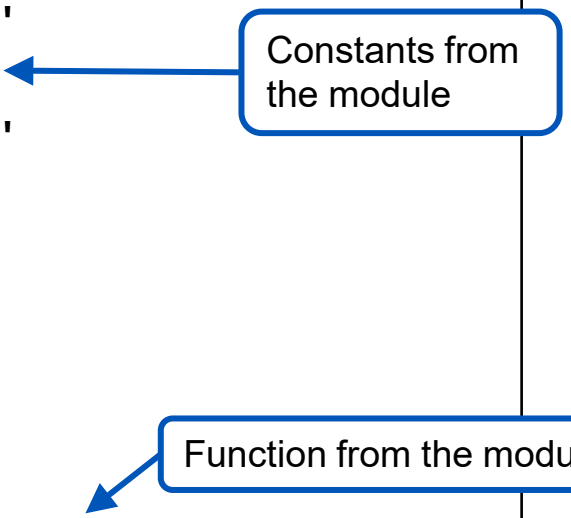

```
$350 0.07 cost =  $ 374.50
```

The string Module

References

- Standard library module providing string functions and constants

```
>>> import string
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.digits
'0123456789'
>>> string.hexdigits
'0123456789abcdefABCDEF'
>>> string.octdigits
'01234567'
>>> string.capwords('now is the time')
'Now Is The Time'
>>> string.capwords('now_is_the_time','_')
'Now_Is_The_Time'
```



Special Strings

- **String objects may contain escape sequences**
  - Special byte encodings that are described following a backslash, `\`

`\', \", \\`	Literal single quote, double quote, backslash
`\r, \n`	Carriage return, newline
`\t`	Tab
`\0num, \xnum`	Character value represented in octal or hexadecimal

- **Raw strings ignore the special meaning of the backslash, `\`**
  - Specified with `r` before the opening quotation mark

```
>>> print('\t is tab')
        is tab
>>> print(r'\t is tab')
\t is tab
```

Keyboard Input

- **The `input('prompt')` function returns one line from standard input**
  - Converted into a string with `\n` removed

strings.py

```
years = input("Enter your age in years --> ")
ageindays = 365.25 * float(years)
print('Your age in days is {a:6.0f}'.format(a=ageindays))
```

Prompt

```
Enter your age in years --> 42
Your age in days is  15340
```

Input

Contents

- Objects and Variables
- Numeric Types and Operations
- String Types and Operations

Conditionals

Simple Comparisons

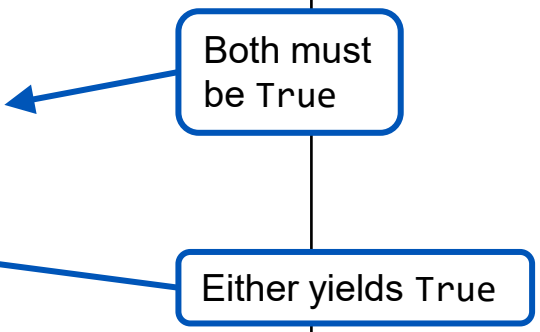
- **Yield a Boolean `True` or `False` value**
- **Types of conditional expressions**
  - Object identity, `is`
  - Arithmetic relational; e.g., `>` or `==`
  - Strings use the same equality and inequality operators as numeric objects

```
>>> sea = 'Atlantic'
>>> ocean = sea
>>> ocean == sea
True
>>> ocean is sea
True
>>> 4 == 4.0
True
>>> 4 is 4.0
False
```

Compound Comparisons

- **Several simple conditions joined by Boolean operators**
  - `and` yields `True` if both operands are `True`
  - `or` yields `True` if either is `True`
  - `not` reverses the Boolean value

```
>>> first = 1
>>> second = 2
>>> third = 3
>>> first < second and second == third
False
>>> first < second or second == third
True
>>> second is third
False
>>> second is not third
True
```

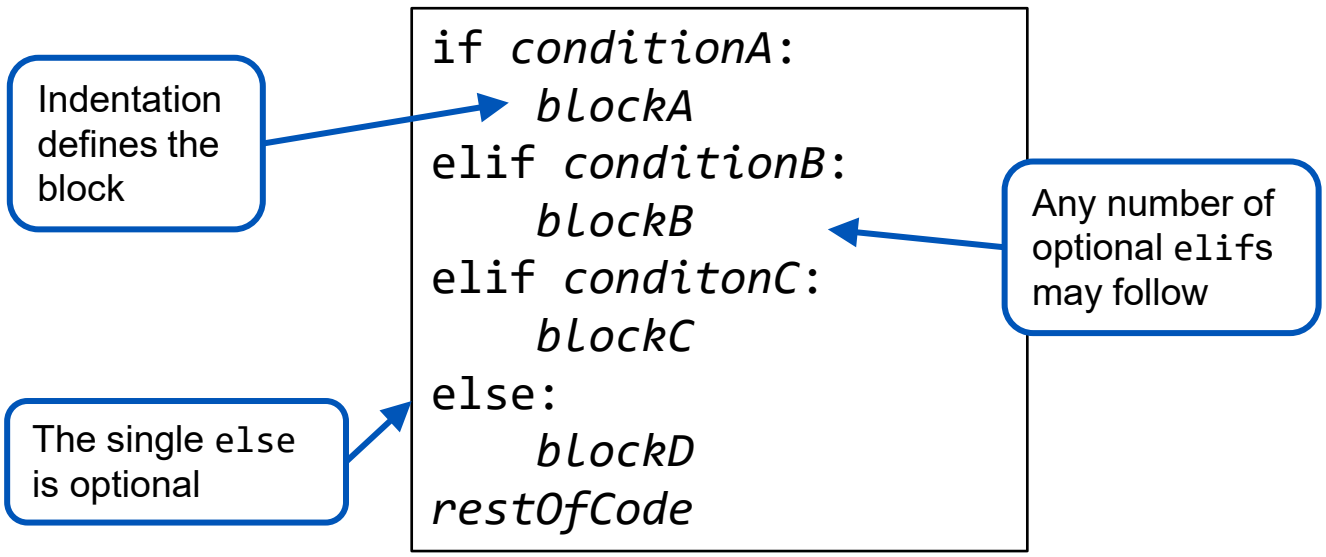


Compound Statements

- **Begin with a header statement that is terminated by a colon, :**
- **Followed by a group of statements that are syntactically treated as a unit—a *suite***
  - A code block
  - For example: a loop body
- **Following statements are tied to the header based on the same indentation**
  - One of Python's readability features
  - Python Enhancement Proposal (PEP) 8 recommends indentation of four spaces
- **When entering a compound statement into the interpreter**
  - Enter an empty line to terminate and run the compound statement

The `if` Statement

- **Evaluates an expression's Boolean value and executes the associated block**
  - `False` is `0`, empty string, empty collection, and `None`; anything else is considered `True`
- **Syntax:**



- **The block associated with first condition that yields `True` is executed**
  - The `else:` block is executed if no condition yields `True`

Simple `if` Testing

conditionals.py

```
sea = 'atlantic ocean'
if sea.islower():
    print(sea.upper())
else:
    print(sea)

sea = input('Enter the sea name: ')
if sea != '' :
    print('sea is not empty')
else:
    print('sea is empty')
```

Yields a Boolean

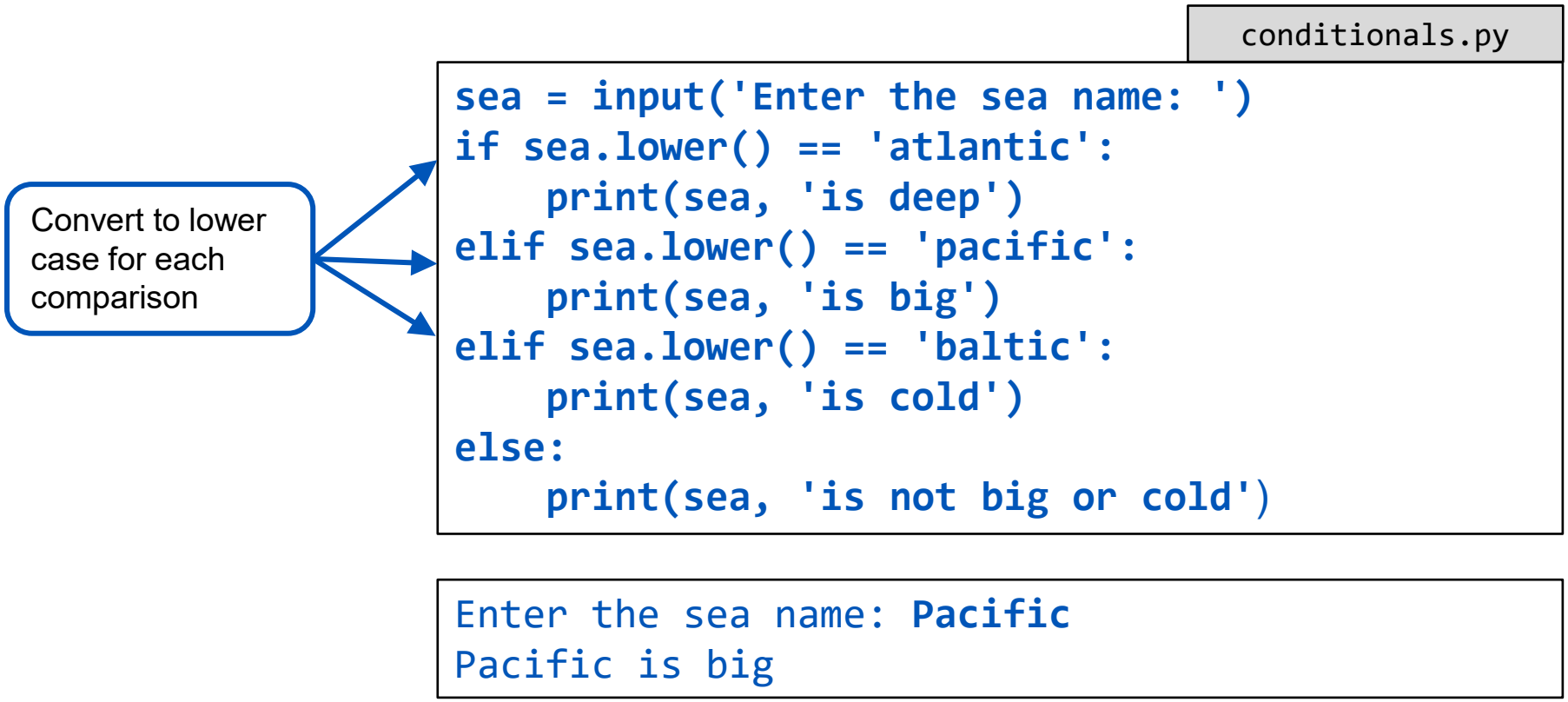
Test for non-empty string

```
ATLANTIC OCEAN
Enter the sea name:
sea is empty
```

Press <Enter> only

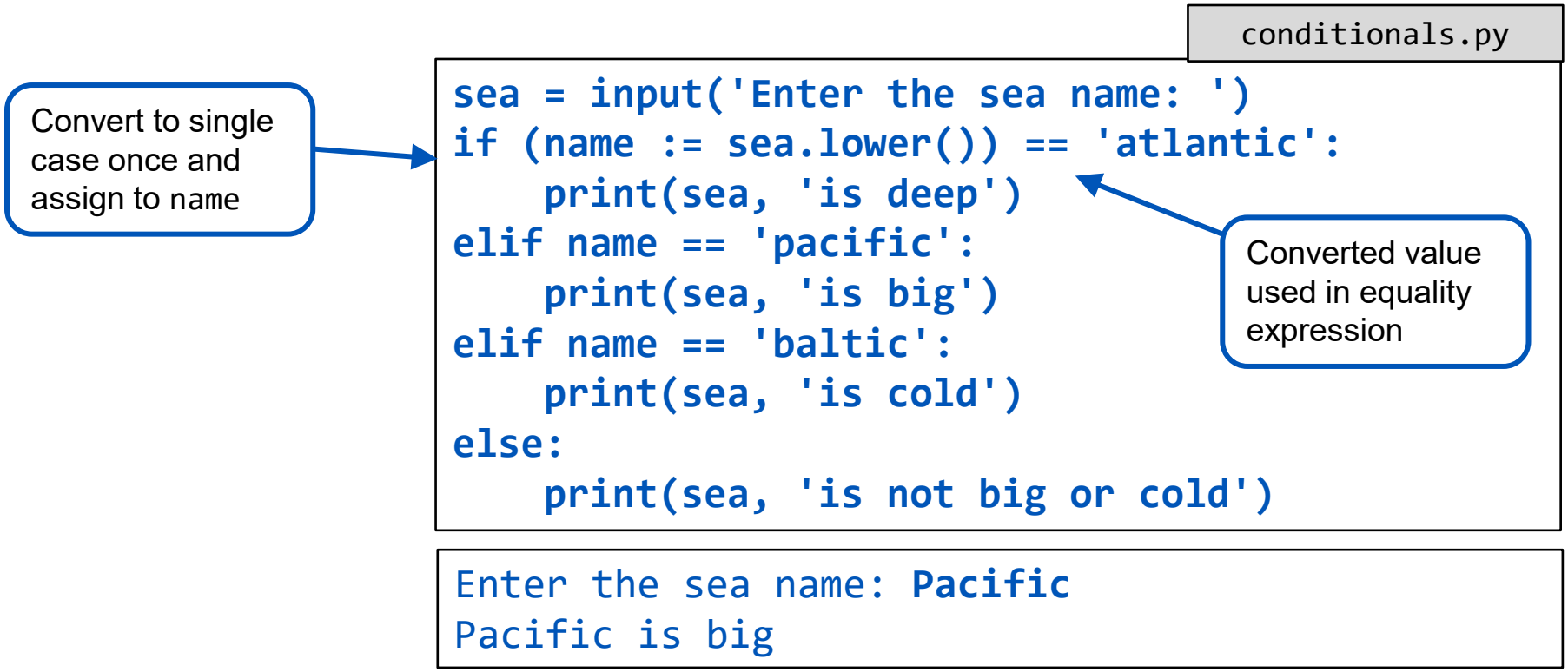
Testing Alternatives Using `elif`

- **A series of tests may be combined using `elif`**



Assignment Expressions

- **The `:=` operator assigns values as part of a larger expression**
  - Not a replacement for assignment statement
  - Values are available in the expression



The pass Statement

- **Explicitly does nothing**
  - Null statement
- **Serves as a placeholder where a statement is required**

conditionals.py

```
sea = input('Enter the sea name: ')
if (name := sea.lower()) == 'atlantic':
    print(sea, 'is cold')
elif name == 'pacific':
    pass
elif name == 'baltic':
    print(sea, 'is cold')
else:
    print(sea, 'is not big or cold')
```


```
Enter the sea name: Pacific
```

Explicit Line Continuation

- **A line ending in a backslash is joined with the following line, forming a single logical line**
  - The backslash and the following end-of-line character are deleted
  - A line ending in a backslash cannot carry a comment.
  - A backslash does not continue a token except for string literals

conditionals.py

```
if sea == 'Atlantic' or \
    sea == 'Pacific' or \
    sea == 'Indian':
    print(sea, 'is a well known sea')
else:
    print('Unknown sea')
```

Implicit Line Continuation

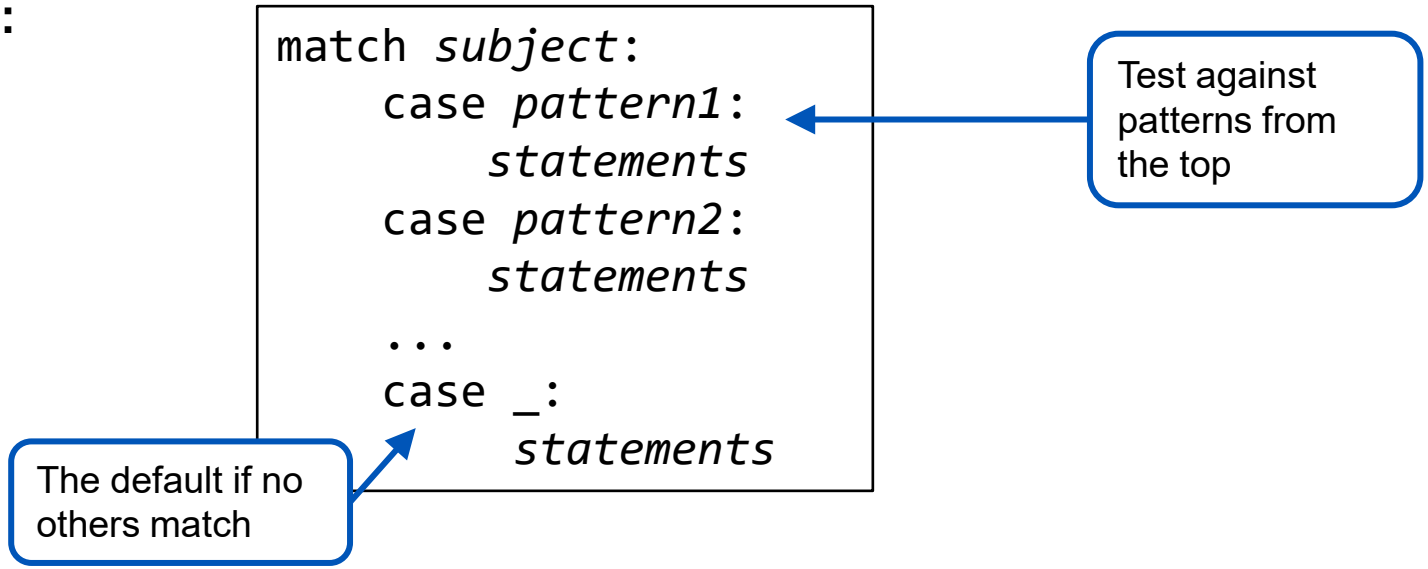
- **Expressions in parentheses, square brackets, or curly braces can be split over more than one physical line without using backslashes**
  - Function argument lists can be separated after the comma

conditionals.py

```
if (sea == 'Atlantic' or
        sea == 'Pacific' or
        sea == 'Indian'):
    print(sea,
          'is a well known sea')
else:
    print('Unknown sea')
```

The `match` Statement

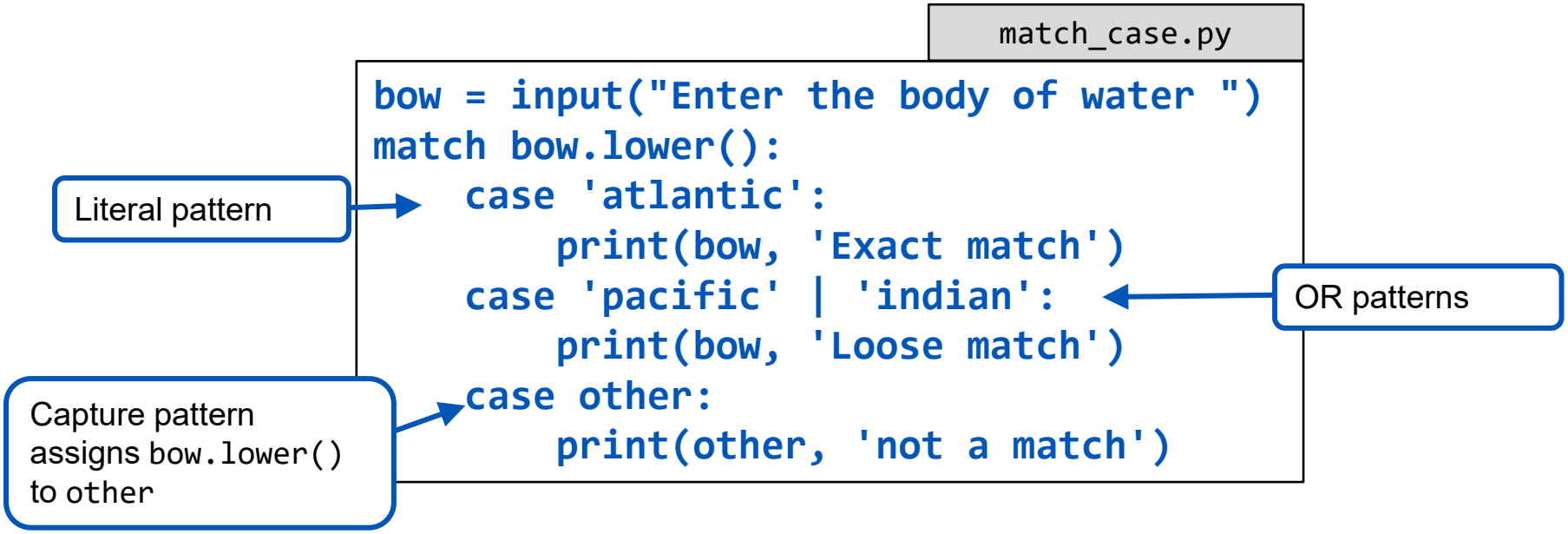
- **Evaluates the *`subject`* value to compare**
  - Attempts to match top down for each pattern
- **Syntax:**



- **The *`statements`* associated with the matched pattern are executed**

The `match` Pattern Types

- **Literal Patterns**
  - Match a string, numeric, or Boolean value
- **OR Patterns**
  - Match either pattern
- **Capture Patterns**
  - Match and bind *`subject`* value to a name



match Example

- **Patterns can also be collection types, like lists**
  - Contains pattern strings and variables
    - Assigned by position

match_case.py

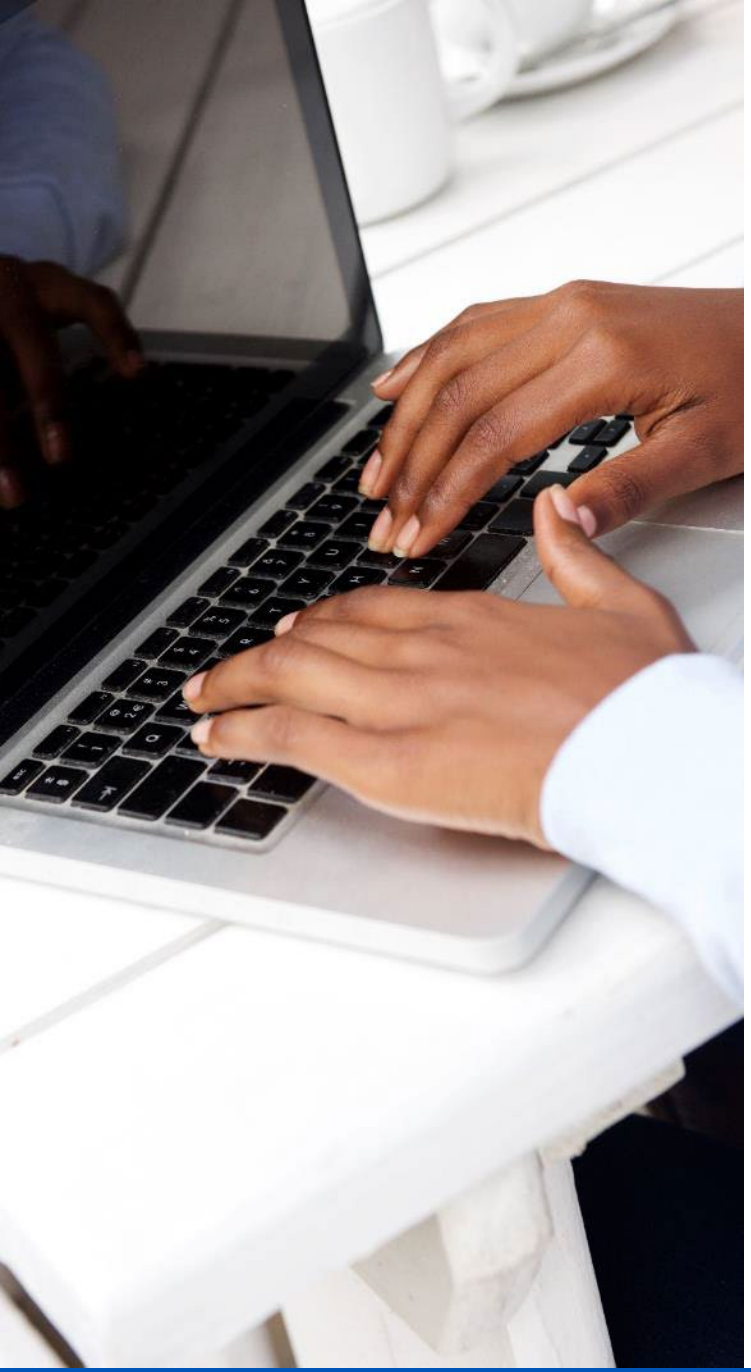
```
bow = input("Enter the body of water ")
bow = bow.lower()
match name := bow.split():
    case [ocean_name, 'ocean'] | [ocean_name, 'sea']:
        print(ocean_name, 'is large')
    case ['gulf', 'of', gulf_name]:
        print(gulf_name, 'is warm')
    case ['lake', lake_name]:
        print(lake_name, 'is fresh water')
    case [river_name, 'river']:
        print(river_name, 'flows downhill')
    case _:
        print(' '.join(name), 'is unknown')
```

Split string into list of strings

List patterns including variables to be assigned

Assigns to `river_name`

Join list of strings to single string



Hands-On Exercise 2.2

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 2.2: Strings and `if`

Objectives

- **Write simple Python programs**
  - Create variables
  - Manipulate numeric values
  - Manipulate string values
  - Make decisions

Chapter 3

Collections

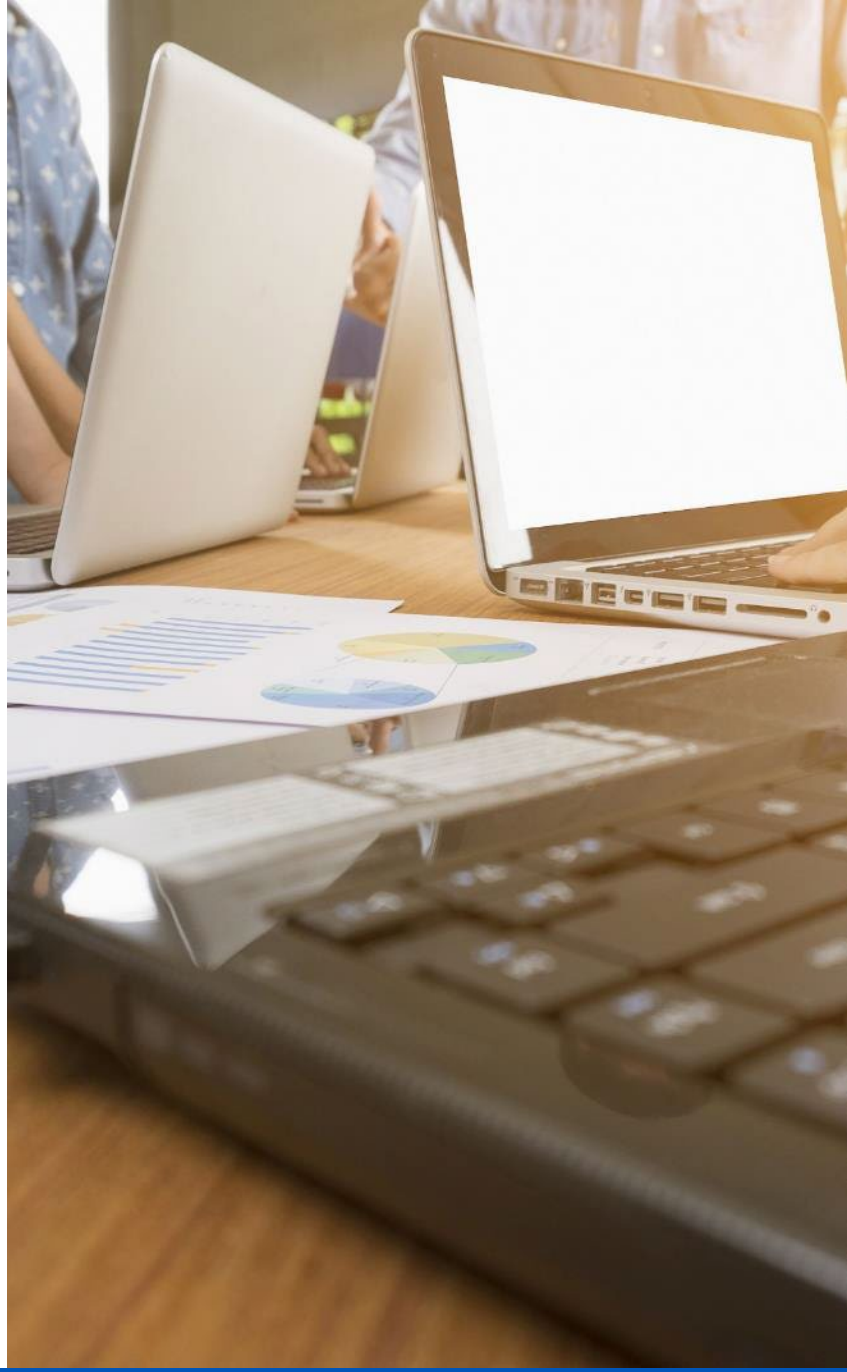
Objectives

- **Create and manage collections**
  - Lists, tuples, sets, and dictionaries
- **Perform iteration**

Contents

Lists, Dictionaries, and Tuples

- `for` Loops and Iterables
- `while` Loops

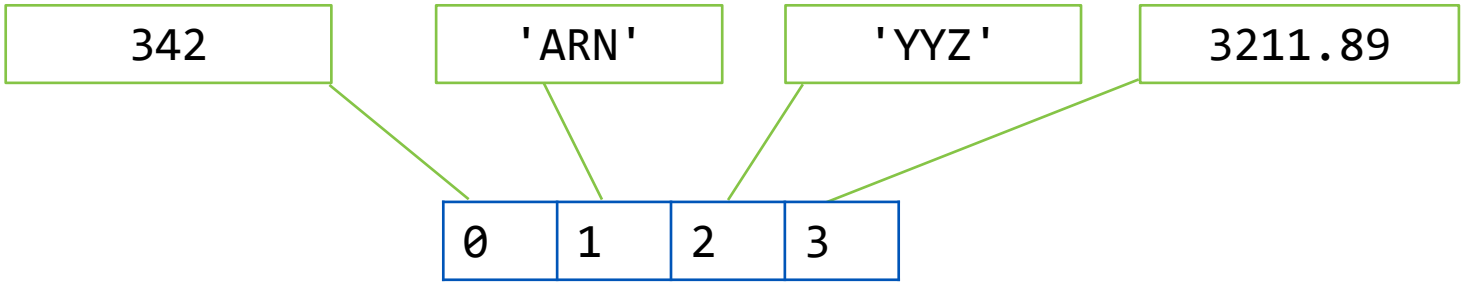


Collections

- **Python provides several types of collections**
  - Compound data types or data structures
    - Composed of elements of various types
- **Collections are categorized as**
  1. Sequential
     - Access individual values by a numeric offset
       - Strings, lists, and tuples
  2. Mapped or associative
     - Access individual values by a key
       - Dictionaries
  3. Unordered
     - Sets
- **May be mutable or immutable**
  - Lists, sets, and dictionary values are mutable
  - Strings, tuples, and dictionary keys are immutable

List

- **Core Python type**
  - Similar to an array in other languages
  - No maximum size
- **Contents may be a combination of different types**
  - Numeric literals, string literals, Booleans, and any other type
- **Represented as a comma-delimited series of values within brackets [ ]**
  - `[342, 'ARN', 'YYZ', 3211.89]`



List Indexing

- **Lists can be assigned**
  - A sequence of values within [ ]
  - Or simply [ ] to represent an empty list
- **Contents are accessed with syntax similar to strings**
  - Numeric offset range described in [ ]

lists_and_tuples.py

```
airports = ['LAX', 'HNL', 'YYZ', 'NRT', 'cdg']
print(airports[1])
airports[4] = airports[4].upper()
print(airports)
```


```
HNL
['LAX', 'HNL', 'YYZ', 'NRT', 'CDG']
```

List Slicing

- **Consecutive elements can be referenced as a slice**
  - [*start*:*end*] syntax as with strings
    - [*start*:*end*:*step*] syntax references every *step*th element in the slice
- **Slice of a list is itself a list**

lists_and_tuples.py

```
airports = ['LAX', 'HNL', 'YYZ', 'NRT', 'CDG']
print(airports[1:2])
print(airports[3:])
print(airports[:])
print(airports[::-1])
```

```
['HNL']
['NRT', 'CDG']
['LAX', 'HNL', 'YYZ', 'NRT', 'CDG']
['CDG', 'NRT', 'YYZ', 'HNL', 'LAX']
```

List Operators

- The + operator concatenates lists

lists_and_tuples.py

```
north_airports = ['YYZ', 'ARN', 'LHR']
south_airports = ['SYD', 'RIO', 'CPT']
print(north_airports + south_airports)
```

```
['YYZ', 'ARN', 'LHR', 'SYD', 'RIO', 'CPT']
```

List Operations

- The `len()` function returns the number of elements in the list
- The `list(arg)` function returns its argument as a list

lists_and_tuples.py

```
airports = ['LAX', 'HNL', 'YYZ', 'NRT', 'CDG']
print(len(airports))
destinations = airports
if destinations is airports:
    print('Shared Reference')
    destinations[0] = 'SFO'
    print('airports changed', airports)
destinations = list(airports)
if destinations is not airports:
    print('Not Shared Reference')
```

Assignment creates shared reference

List is copied

```
5
Shared Reference
airports changed ['SFO', 'HNL', 'YYZ', 'NRT', 'CDG']
Not Shared Reference
```

List Modifications

- **Assignment modifies the list in place**
- **Method functions allow in-place modification of list contents**
  - *list*.append(*value*)—Add *value* to the end
  - *list*.pop(*n*)—Remove element *n* and return it
  - *list*.insert(*posit*,*value*)—Add *value* at position *posit*
  - *list*.sort() and *list*.reverse()—Change contents sequence

```
airports = ['LAX', 'HNL', 'YYZ', 'NRT', 'CDG']
airports[0] = 'LGA'
airports[2:3] = ['MSY', 'SFO']
print(airports)
airports.append('JFK')
airports.insert(0, 'SYD')
print(airports)
airports.sort()
print(airports)
```

lists_and_tuples.py

```
['LGA', 'HNL', 'MSY', 'SFO', 'NRT', 'CDG']
['SYD', 'LGA', 'HNL', 'MSY', 'SFO', 'NRT', 'CDG', 'JFK']
['CDG', 'HNL', 'JFK', 'LGA', 'MSY', 'NRT', 'SFO', 'SYD']
```

Tuple

- **A sequenced type**
  - Contents are accessed by an offset like a list
- **An immutable type**
  - No change in size or content after creation
- **Normally represented by a comma-delimited list of values within ( )**

```
>>> airports = ('LAX', 'HNL', 'YYZ', 'NRT', 'CDG')
>>> airports[1]
'HNL'
>>> airports[1] = 'LNY'

Traceback (most recent call last):
TypeError: 'tuple' object does not support item assignment
```

Tuple

- Parentheses are optional on assignment
- Single element tuple requires a comma on assignment
- `()` creates an empty tuple

```
>>> planes = 'A350', 'A380', 'B747', 'B737'
>>> planes
('A350', 'A380', 'B747', 'B737')
>>> biggest_plane = ('A380',)
>>> biggest_plane
('A380',)
```

- What type of object would `start = (1)` create?

Tuple Operations

- **Consecutive elements are accessed with standard slice notation**
  - Slice of a tuple is itself a tuple
- **+ and * operators concatenate or repeat tuples**
- **The `tuple()` function returns its argument as a tuple**

lists_and_tuples.py

```
airports = ('LAX', 'HNL', 'YYZ', 'NRT', 'CDG')
print(airports[1:3])
print(airports[::-1])
north_airports = ['YYZ', 'ARN', 'LHS']
south_airports = ['SYD', 'RIO', 'CPT']
destinations = tuple(north_airports + south_airports)
print(destinations)
```

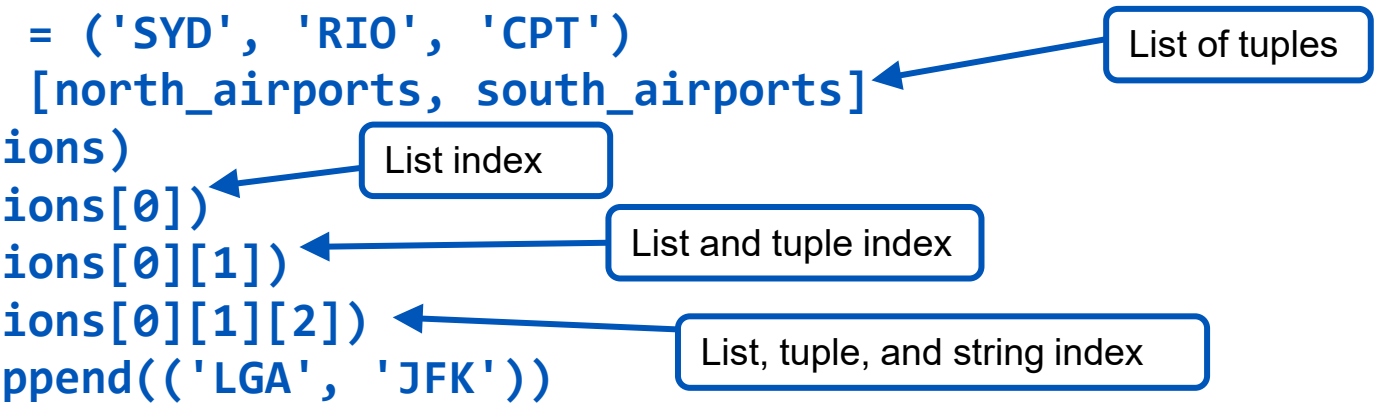
```
('HNL', 'YYZ')
('CDG', 'NRT', 'YYZ', 'HNL', 'LAX')
('YYZ', 'ARN', 'LHS', 'SYD', 'RIO', 'CPT')
```

Collections of Collections

- **Lists and tuples may contain any type of object**
  - Including lists and tuples

lists_and_tuples.py

```
north_airports = ('YYZ', 'ARN', 'LHS')
south_airports = ('SYD', 'RIO', 'CPT')
destinations = [north_airports, south_airports]
print(destinations)
print(destinations[0])
print(destinations[0][1])
print(destinations[0][1][2])
destinations.append(('LGA', 'JFK'))
print(destinations)
```




```
[('YYZ', 'ARN', 'LHS'), ('SYD', 'RIO', 'CPT')]
('YYZ', 'ARN', 'LHS')
ARN
N
[('YYZ', 'ARN', 'LHS'), ('SYD', 'RIO', 'CPT'), ('LGA', 'JFK')]
```

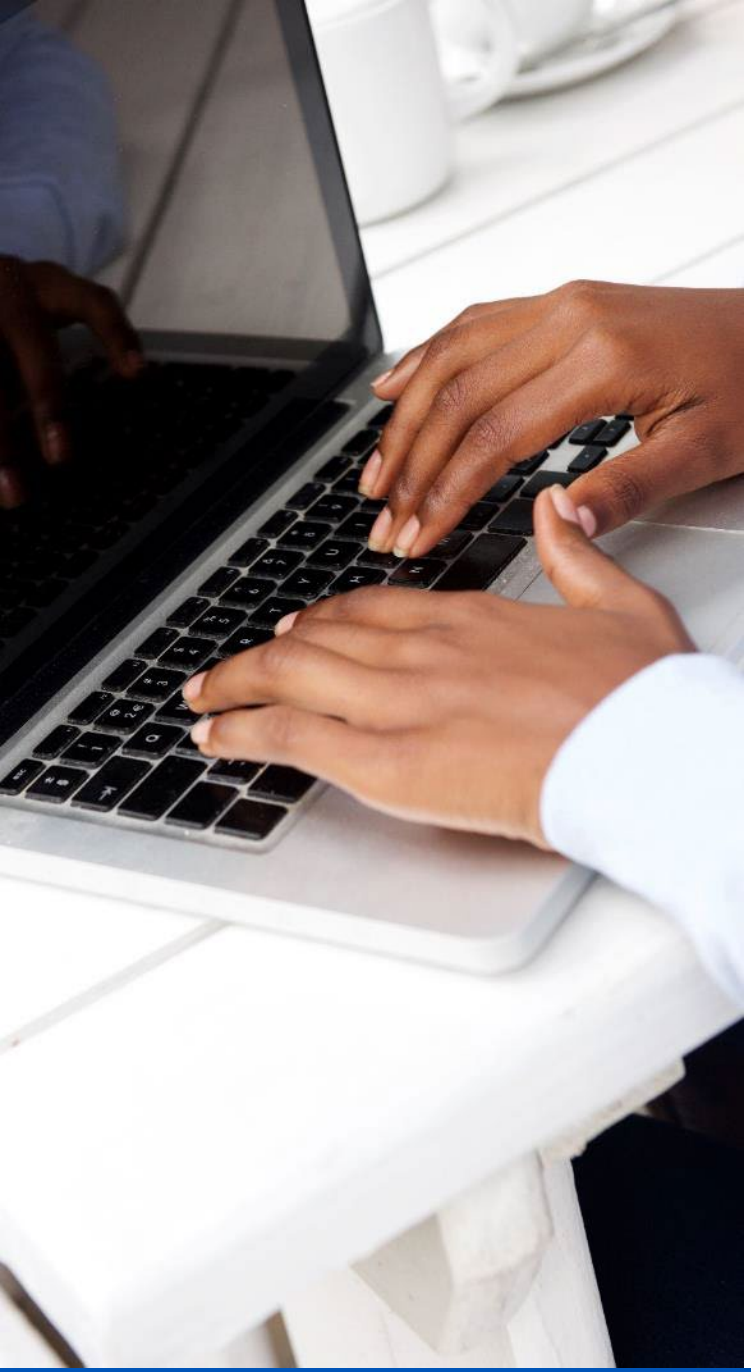
Sequence Unpacking

- **Multiple values from a collection are assigned by position**
  - Collection slice is allowed
- **Multiple values may be assigned to a wildcard variable**
  - References a list of the remaining values

lists_and_tuples.py

```
airports = ('LAX', 'HNL', 'YYZ', 'NRT')
depart, layover1, layover2, arrive = airports
print(layover2)
depart, *layovers, arrive = airports
print(layovers)
depart, *rest = airports
print(rest)
```

```
YYZ
['HNL', 'YYZ']
['HNL', 'YYZ', 'NRT']
```



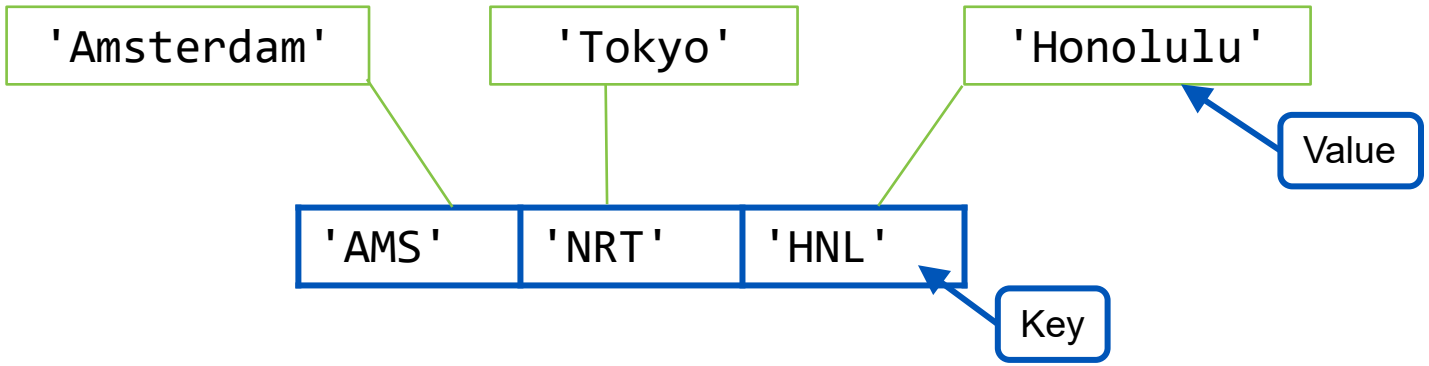
Hands-On Exercise 3.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 3.1: Collections and Slicing

Dictionary

- **Associative type**
  - Contents accessed through symbolic keys instead of numeric indices
    - No maximum size
    - Similar to an associative array or hash in other languages
- **Contents may be composed of any combination of types**
  - Numeric literals, string literals, lists, or any other type
- **Is represented within curly braces { } by a comma-delimited series of `key:value` pairs**
  - `{'AMS': 'Amsterdam', 'NRT': 'Tokyo', 'HNL': 'Honolulu'}`



Dictionary Operations

- **Any individual element can be referenced through its key**
  - Value for that key may be retrieved or updated
- **Keys are unique, immutable objects**
  - Change in key implies a new entry for the dictionary
  - Keys may be numeric literals, strings, or tuple elements
- **`del` statement removes a key–value pair from the dictionary**

```
cities = {'YYZ': 'Toronto', 'NRT': 'Tokyo/Narita'}
print(cities['NRT'])
cities['NRT'] = 'Tokyo'
cities['HNL'] = 'Honolulu'
print(cities)
del cities['YYZ']
print(cities)
```

dictionaries_and_sets.py

```
Tokyo/Narita
{'YYZ': 'Toronto', 'NRT': 'Tokyo', 'HNL': 'Honolulu'}
{'NRT': 'Tokyo', 'HNL': 'Honolulu'}
```

Dictionary Methods and Functions

- **Method functions allow data retrieval or modification of contents**
  - `dict.get(key[,default])`—Returns value at *key*, else *default* for undefined *key*
    - If *default* not provided, return None
  - `len(dict)`—Returns the number of items in the dictionary

dictionaries_and_sets.py

```
depcode = input("Enter the departure ")
print(cities.get(depcode.upper(), "Incorrect code"))
arrcode = input("Enter the arrival ")
print(cities[arrcode.upper()])
```

```
Enter the departure lga
Incorrect code
Enter the arrival lga
Traceback (most recent call last):
...dictionaries_and_sets.py", line 12, in <module>
    print(cities[arrcode.upper()])
KeyError: 'LGA'
```

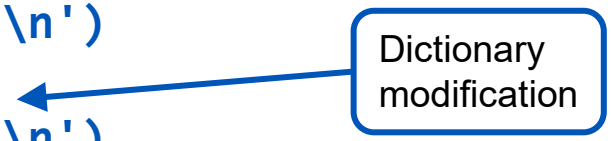
Key Error exception is raised

View Objects

- **Provide a dynamic view of the referenced object**
  - Change as the dictionary object changes
  - *dict*.keys()—Returns keys
  - *dict*.values()—Returns values
  - *dict*.items()—Returns a dictionary view of key–value pairs as tuples

dictionaries_and_sets.py

```
cities = {'YYZ': 'Toronto', 'NRT': 'Tokyo'}
ckeys = cities.keys()
cvalues = cities.values()
print(ckeys, cvalues, sep='\n')
cities['HNL'] = 'Honolulu'
print(ckeys, cvalues, sep='\n')
print(cities.items())
```




```
dict_keys(['YYZ', 'NRT'])
dict_values(['Toronto', 'Tokyo'])
dict_keys(['YYZ', 'NRT', 'HNL'])
dict_values(['Toronto', 'Tokyo', 'Honolulu'])
dict_items([('YYZ', 'Toronto'), ('NRT', 'Tokyo'), ('HNL',
'Honolulu')])
```

Creating a Dictionary

- **A dictionary can be created from a sequence of key–value pairs**
  - `dict(`*`arg`*`)` returns a dictionary
  - *arg* can be a sequence containing the key–value pairs
  - *arg* can be keyword arguments of the key–value pairs
- ***name*** **= {}—Creates an empty dictionary**
  - `newcities = {}`

dictionaries_and_sets.py

```
apts = {'SFO': 'San Francisco', 'LAX': 'Los Angeles'}
print(apts)
apts = dict((('SFO', 'San Francisco'), ('LAX', 'Los Angeles')))
print(apts)
apts = dict(SFO='San Francisco', LAX='Los Angeles')
print(apts)
```

Sequence of pairs

Keyword arguments

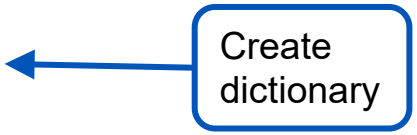
```
{'SFO': 'San Francisco', 'LAX': 'Los Angeles'}
{'SFO': 'San Francisco', 'LAX': 'Los Angeles'}
{'SFO': 'San Francisco', 'LAX': 'Los Angeles'}
```

zip() Function

- **Combines two collections in parallel and returns a new list**
  - Composed of two element tuples based on position
  - Returned list is the length of the shorter argument

dictionaries_and_sets.py

```
codes = ['ORD', 'MCO']
cities = ['Chicago', 'Orlando']
bycodes = dict(zip(codes, cities))
bycities = dict(zip(cities, codes))
print(bycodes)
print(bycities)
```




```
{'ORD': 'Chicago', 'MCO': 'Orlando'}
{'Chicago': 'ORD', 'Orlando': 'MCO'}
```

Copying a Dictionary

- ***dict*.copy()—Returns a shallow copy of the dictionary**
  - Does not copy any embedded collections

dictionaries_and_sets.py

```
cities = {'YYZ': 'Toronto', 'NRT': 'Tokyo'}
favcities = cities
print(favcities is cities)
favcities = cities.copy()
print(favcities is cities)
print(favcities == cities)
```

Assignment creates a shared reference

Equality of value

```
True
False
True
```

Sets

- **Unsequenced mutable collections of unique, immutable objects**
  - Like dictionary keys
  - Created with the `set()` function and sequence argument
  - Or by assignment with `{ }`
- **The *`set`*`.add()` and *`set`*`.remove()` methods can add or remove members**
- **Used to**
  - Remove duplicates
  - Test membership

dictionaries_and_sets.py

```
hawaii_airports = set(['HNL', 'ITO', 'HNL'])
pacific_airports = {'HNL', 'NRT', 'SYD', 'LAX'}
hawaii_airports.add('LNY')
pacific_airports.remove('LAX')
print(hawaii_airports)
print(pacific_airports)
```

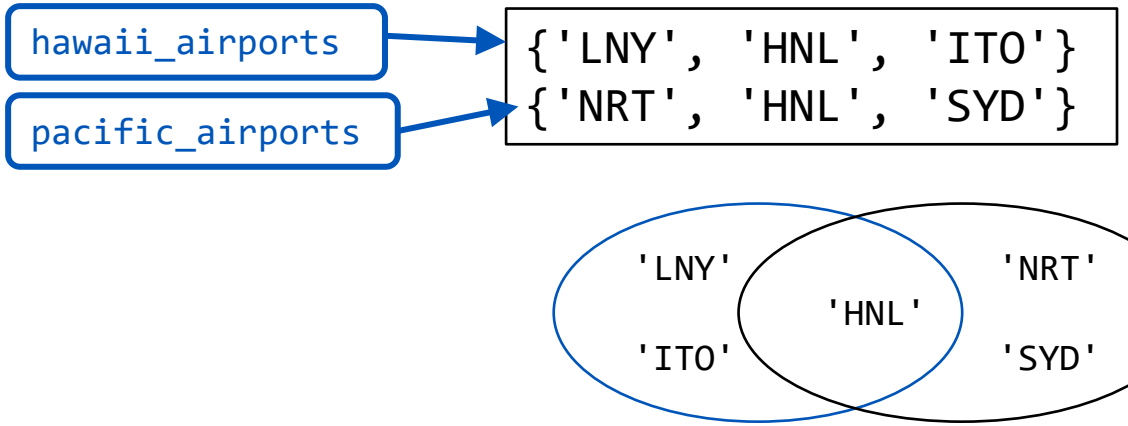
Mutable collection

```
{'LNY', 'HNL', 'ITO'}
{'NRT', 'HNL', 'SYD'}
```

Duplicates removed

Set Operators

-	Difference
|	Union
&	Intersection
>	Superset
<	Subset
==	Equality
!=	Inequality



dictionaries_and_sets.py

```
print(hawaii_airports - pacific_airports)
print(pacific_airports - hawaii_airports)
print(hawaii_airports | pacific_airports)
print(hawaii_airports & pacific_airports)
print(hawaii_airports > pacific_airports)
```

```
{'ITO', 'LNY'}
{'NRT', 'SYD'}
{'NRT', 'LNY', 'SYD', 'HNL', 'ITO'}
{'HNL'}
False
```

Sets Membership Quiz

Do Now

1. Open `Chap3_Examples` folder and the `sets_quiz.py` file in PyCharm
2. Perform the membership testing as described in the comments

```
# Given the following dictionaries:

codes = {'France': 33, 'Japan': 81,
         'GreatBritain': 44, 'USA': 1}
caps = {'France': 'Paris', 'Cuba': 'Havana',
        'Japan': 'Tokyo'}

# Create a list of the keys in the codes dictionary
# that are also keys in the caps dictionary

# Create a list of the keys in the codes dictionary
# that are not keys in the caps dictionary

# HINT 1: Use set operators & and -
# HINT 2: Shortcut, use set operators directly on the keys
```

Collection Membership Testing: `in`

- **Syntax:** ***`value`*** **`in`** ***`collection`***
  - Returns `True` if the value is a member
  - Works for all collection types and strings
    - Dictionary name alone yields keys

```
>>> 'test' in ('Always', 'test', 'your', 'data')
True
>>> 'test' in {'Always', 'test', 'your', 'coding'}
True
>>> facts = {'test': 'Good idea', 'no test': 'Bad idea'}

>>> 'test' in facts
True
>>> 'test' in facts.values()
False
```

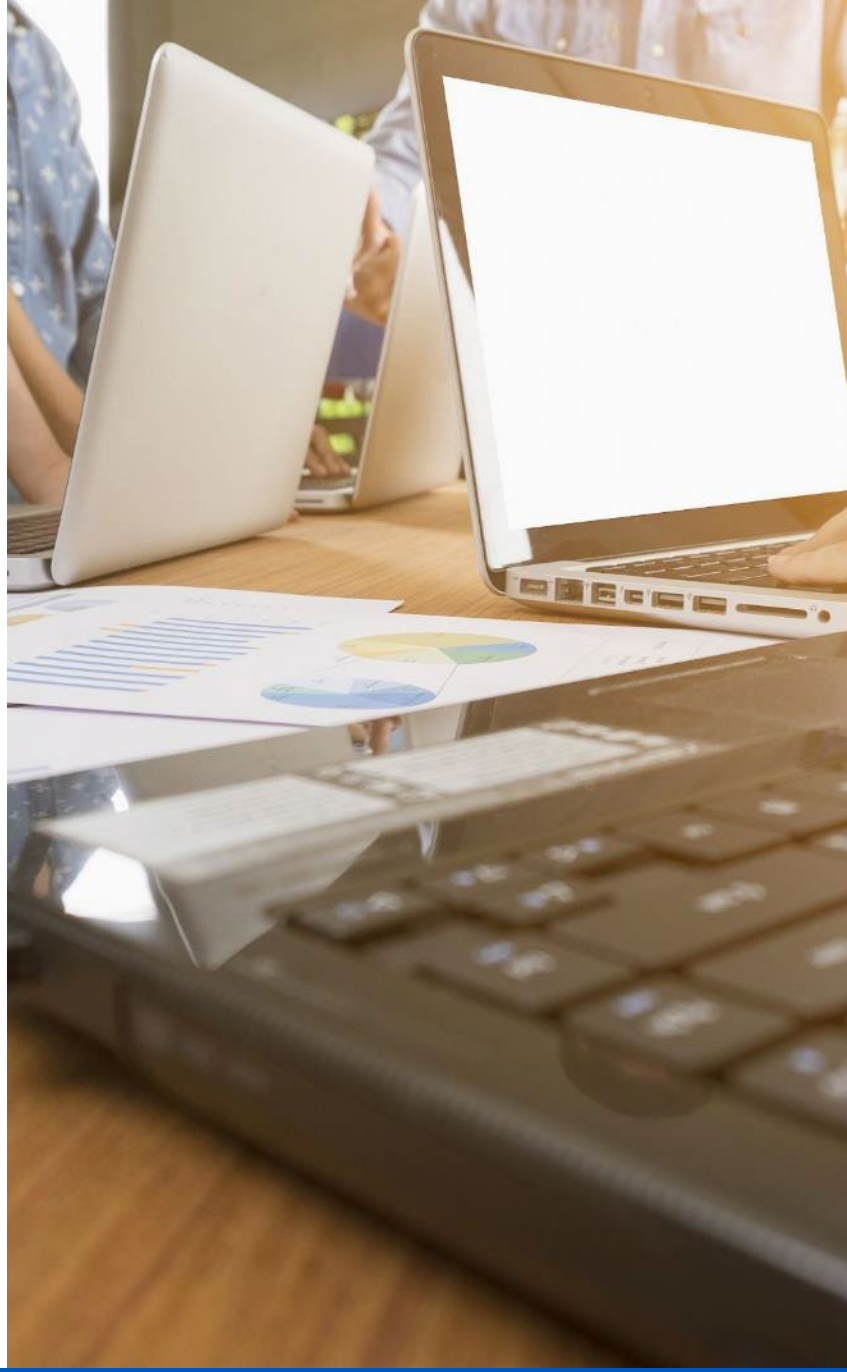
same as `facts.keys()`

Contents

- Lists, Dictionaries, and Tuples

for Loops and Iterables

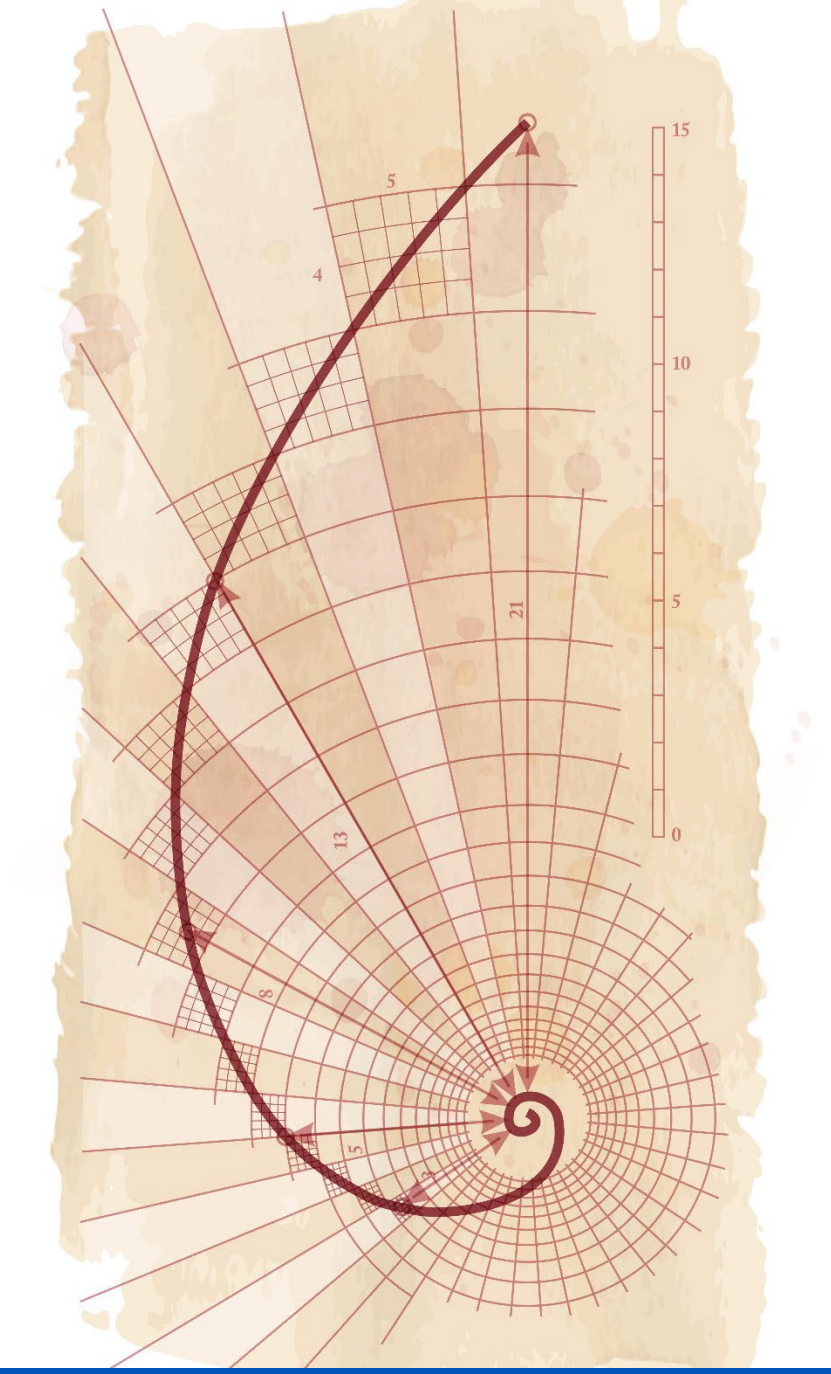
- `while` Loops



Flow Control With Loops

- Fixed number of iterations with `for`
- Conditional iterations with `while`
- Loop body is defined by its indentation

```
Loop statement:
    loopStatement1
    loopStatement2
    ...
restOfCode
```



The for Loop

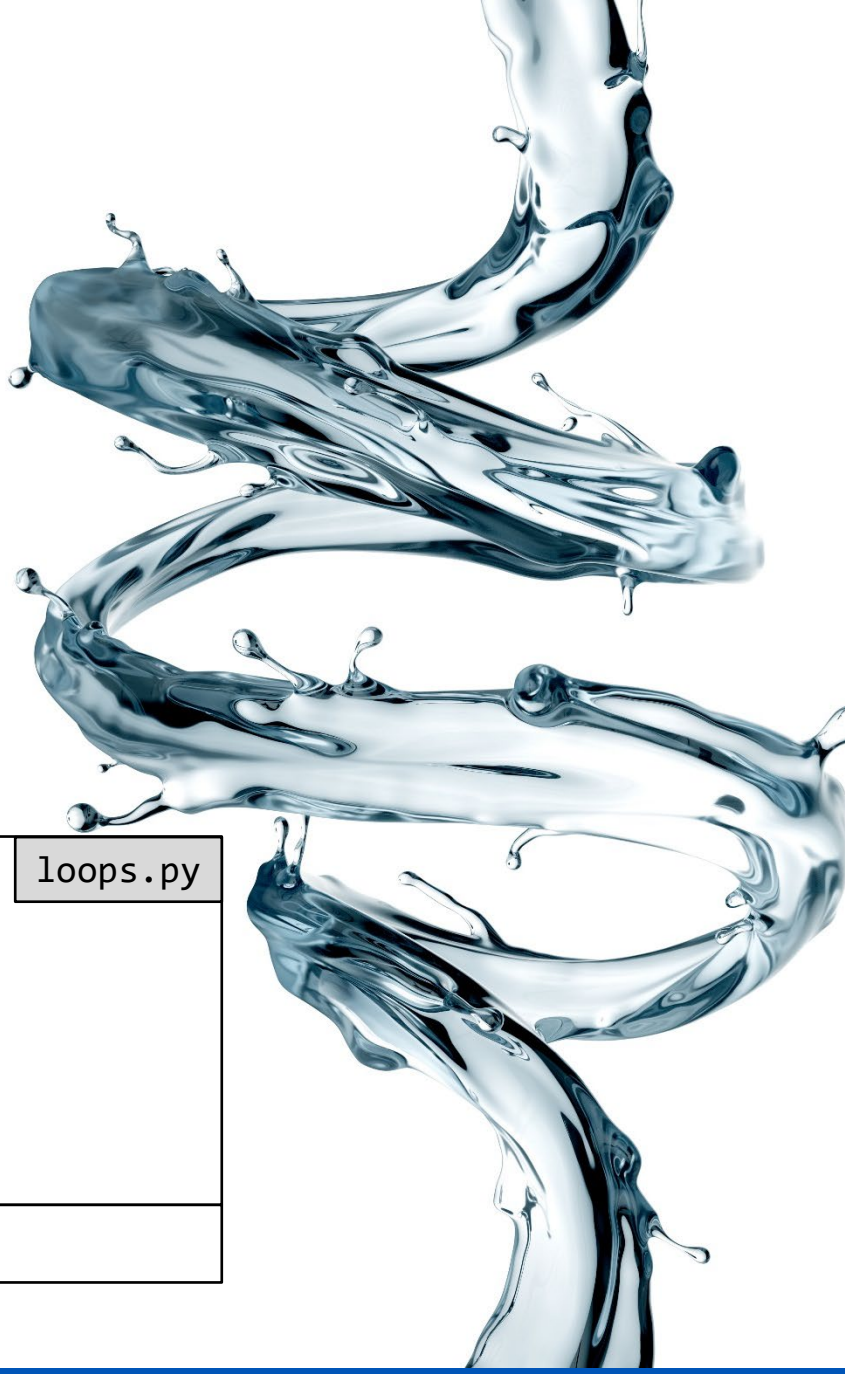
- **Steps through an iterable object**
  - *var* is assigned each object in turn
  - When the iterable is exhausted, exit the loop
- **Syntax:**

```
for var in iterable:
    loopBlock
restOfCode
```

loops.py

```
prices = [200, 400, 500]
fee = 20
totals = []
for price in prices:
    totals.append(price - fee)
print(totals)
```

```
[180, 380, 480]
```



Iterable Objects

- **Object capable of returning its members one at a time**
  - Sequence types: lists, tuples, and strings
  - Nonsequence types: sets, dictionaries, files
- **Can be used when a sequence is needed**
  - `for` loops
  - `zip()` function
- **Iterator**
  - Object representing one pass over a stream of data
  - Can be created using `iter()` function
  - All iterators are iterable

Loop Through a Dictionary

- **Dictionary methods `keys()`, `values()`, and `items()` can provide an iterable object**
  - Dictionary name alone provides the keys

loops.py

```
airports = {'YYZ': 'Toronto', 'NRT': 'Tokyo'}

for code in airports.keys():
    print(code)

for code in airports:
    print(code)

for value in airports.values():
    print(value)

for key, value in airports.items():
    print(key, value)
```

Both print
YYZ
NRT

Toronto
Tokyo

YYZ Toronto
NRT Tokyo

Nested Looping

loops.py

```
prices = [200, 400, 500]
fees = [20, 50]
totals = []
for fee in fees:
    for price in prices:
        totals.append(price - fee)
print(totals)
```

```
[180, 380, 480, 150, 350, 450]
```

Membership Quiz With a Loop

- Create a list of keys from codes that are also keys in caps

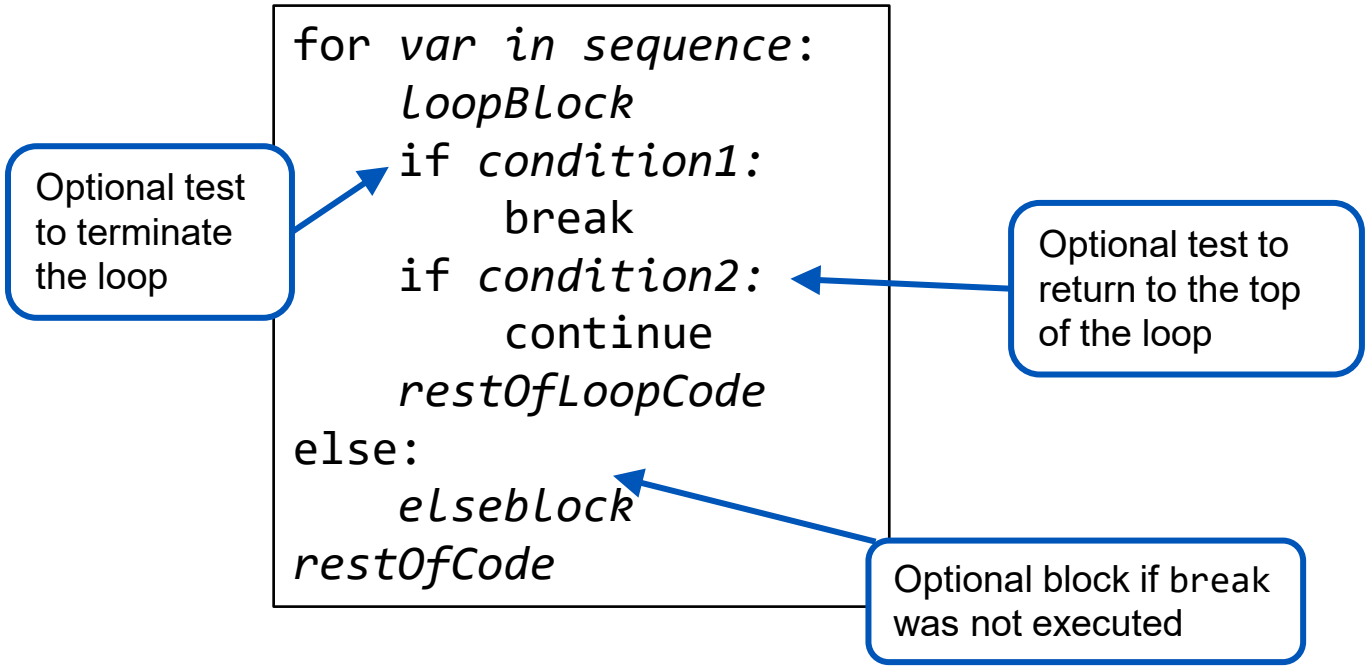
loops.py

```
codes = {'France': 33, 'Japan': 81,
         'GreatBritain': 44, 'USA': 1}
caps = {'France': 'Paris', 'Cuba': 'Havana',
        'Japan': 'Tokyo'}

countries = []
for code in codes:
    if code in caps:
        countries.append(code)

print(countries)
```

```
['France', 'Japan']
```

Optional Flow Control Within Loops

- **`break` terminates the loop**
- **`continue` returns flow control to the top of the loop**
- **`else:` defines a block of code executed after the loop terminates normally**
  - Without a `break`



Using break, continue, and else in a Loop

loops.py

```
airports = ['LAX', 'HNL', 'YYZ']
for airport in airports:
    if airport == 'HNL':
        break
    print('with break', airport)
else:
    print('The end', airport)

for airport in airports:
    if airport == 'HNL':
        continue
    print('with continue', airport)
```

Terminate with `'HNL'`

Skip with `'HNL'`

```
with break LAX
with continue LAX
with continue YYZ
```

The range Class

- **Defines an iterable object that provides a sequence of integers**
  - Commonly used in `for` loops
  - Much less memory than an equivalent list or tuple
- **Syntax:**
  - `range(`*`start`*`,`*`end`*`,`*`step`*`)`
    - *`start`*—starting value, default is `0`
    - *`end`*—noninclusive stopping value, required
    - *`step`*—stepping value, default is 1

Negative step used for decreasing

```
>>> for index in range(3):
...     print(index)
```

Displays:
0
1
2

```
>>> for index in range(5,2,-1):
...     print(index)
```

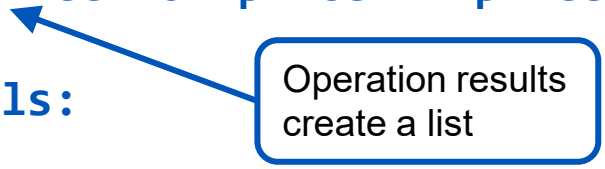
Displays:
5
4
3

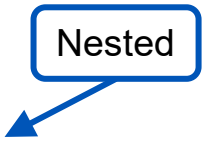
List Comprehension

- Process parts of a sequence and return a list of results
- An *operation* is applied to each element with a `for` loop
- Syntax: [*operation* `for` *var* `in` *iterable*]

comprehensions.py

```
prices = [200, 400, 500]
fee = 20
totals = [price - fee for price in prices]
print(totals[0])
for total in totals:
    print(total)

fees = [20, 50]
print([price - fee for fee in fees for price in prices])
```






```
180
180
380
480
[180, 380, 480, 150, 350, 450]
```

List Comprehension With Conditional

- **The *operation* may be executed conditionally**
  - With an embedded `if`

```
[operation for var in iterable if condition]
```

comprehensions.py

```
fee = 30
minimum = 200
print([price - fee for price in prices if price > minimum])

codes = {'France': 33, 'Japan': 81,
         'GreatBritain': 44, 'USA': 1}
caps = {'France': 'Paris', 'Cuba': 'Havana',
        'Japan': 'Tokyo'}

print([code for code in codes if code in caps])
```

List of keys from `codes` that are also keys in `caps`

```
[370, 470]
['France', 'Japan']
```

Additional Comprehensions

- **Set comprehensions**

```
{operation for var in set if condition}
```

```
airports = {'LAX', 'HNL', 'YYZ'}
hawaiiairports = {airport for airport in airports
                  if airport in ['HNL', 'ITO']}
print(hawaiiairports)
```

```
{'HNL'}
```

Result is a set

- **Dictionary comprehensions**

```
{key: value for key, value in sequence if condition}
```

```
airports = {'LAX': 'Los Angeles', 'HNL': 'Honolulu',
            'YYZ': 'Toronto'}
hawaiidict = {code: city for code, city in airports.items()
              if code in ['HNL', 'ITO']}
print(hawaiidict)
```

```
{'HNL': 'Honolulu'}
```

Result is a dictionary

Generator Expression

- **Creates an iterator that supports the `next()` method**
- **The *`operation`* is applied to each element with a `for` loop when requested using `next()`**
  - No data structure of all results created
- **Syntax: (*`operation`* `for` *`var`* `in` *`iterable`*)**

comprehensions.py

```
prices = [200, 400, 500]
fee = 20
totals = (price - fee for price in prices)
print(next(totals))
print('start loop')
for total in totals:
    print(total)
```

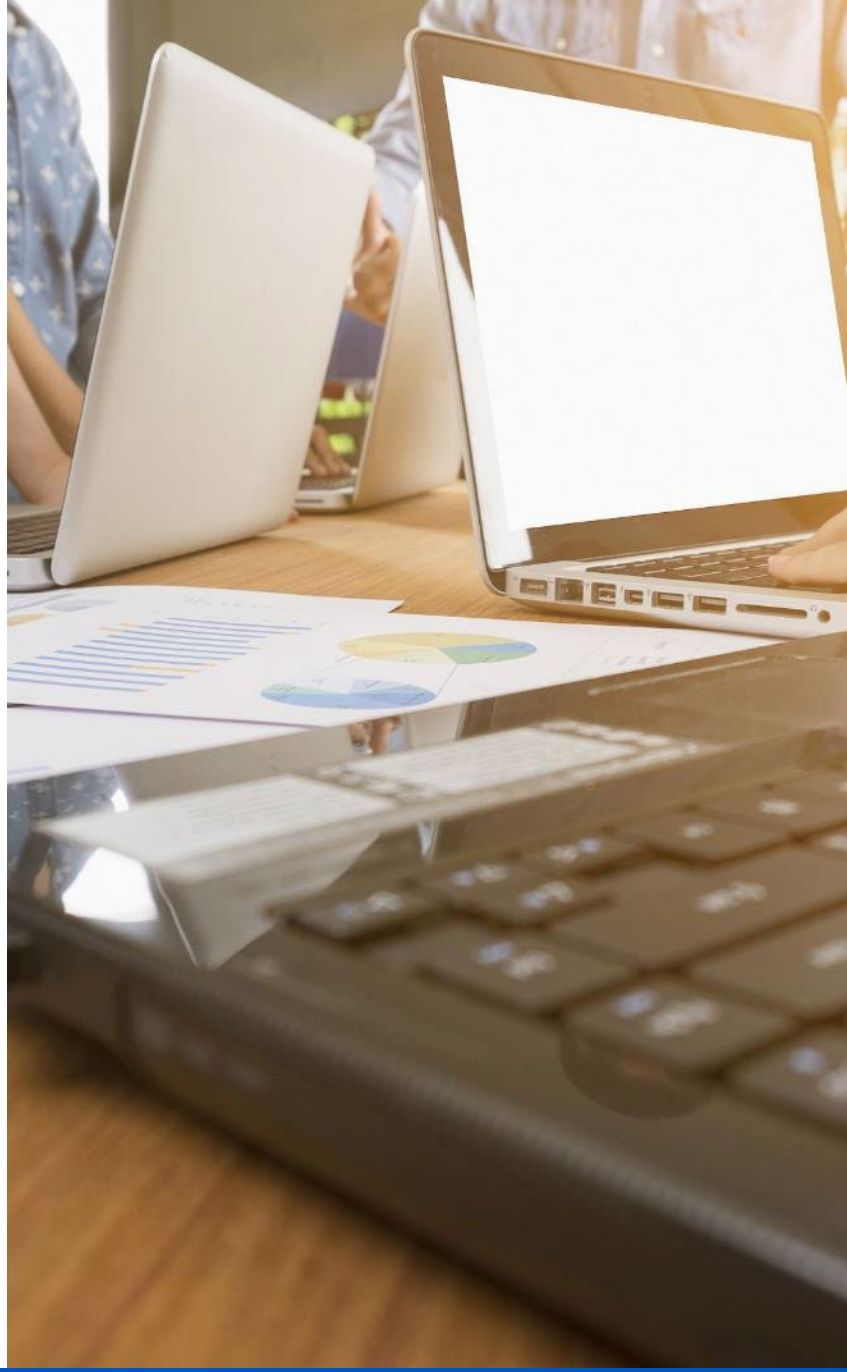
Result is a generator object

```
180
start loop
380
480
```

Contents

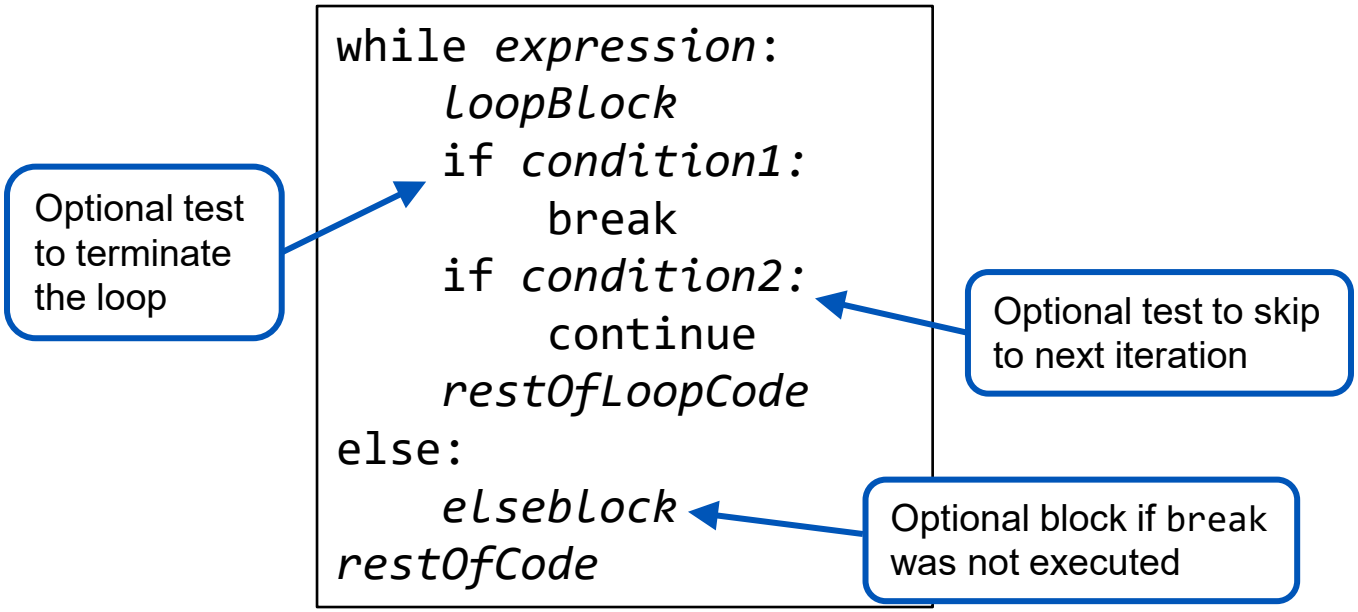
- Lists, Dictionaries, and Tuples
- `for` Loops and Iterables

`while` Loops



The `while` Loop

- **Evaluates the Boolean value of an expression**
  - Executes the loop body so long as the expression is `True`
    - Terminates on `False` or execution of a `break`
- **Syntax:**



`while` Loop Example

Assign input to guess

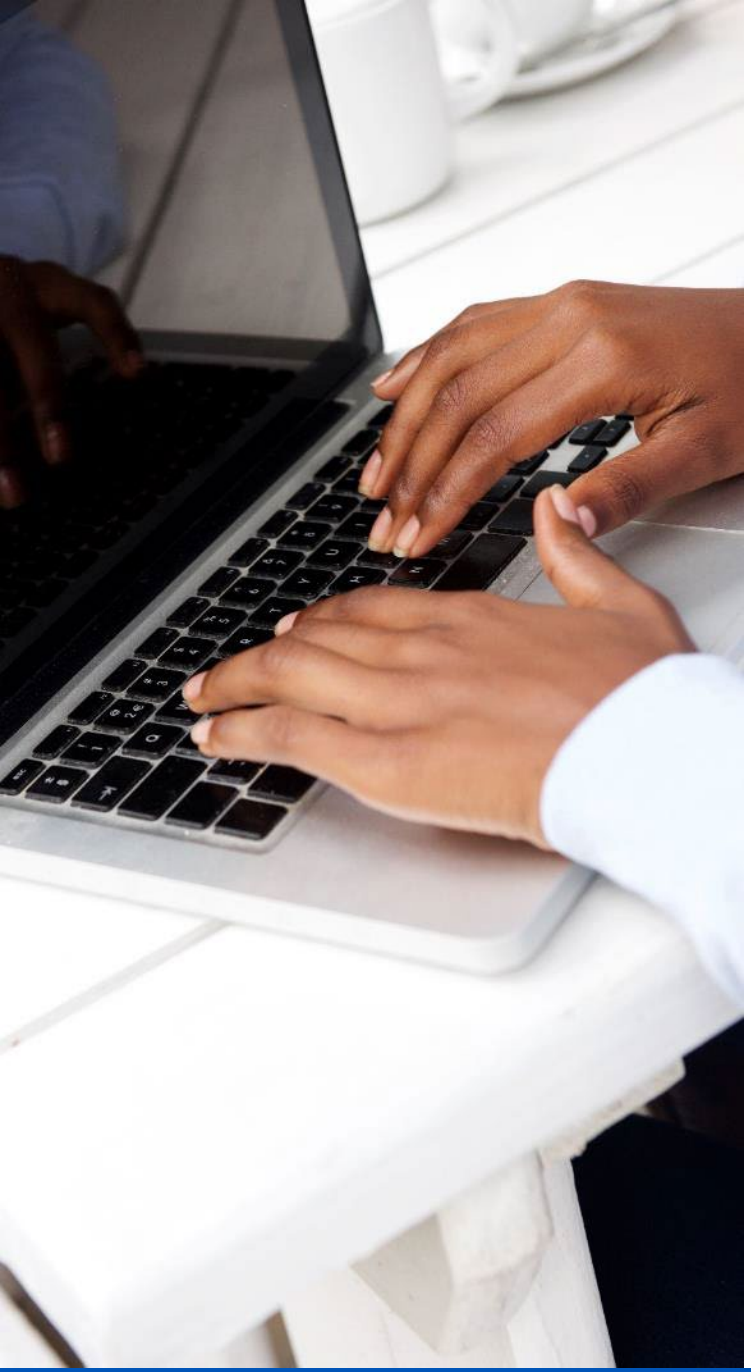
query_loop.py

```
creator = 'Guido'
while (guess := input('Who is the creator of Python? ')) != creator:
    print('Sorry but', guess, 'is not correct.  Try again')
else:
    print('Right, it is', creator)
```

Compare guess to creator

Executes at termination of `while`

```
Who is the creator of Python? Linus
Sorry, but Linus is not correct.  Try again
Who is the creator of Python? Larry
Sorry, but Larry is not correct.  Try again
Who is the creator of Python? Guido
Right, it is Guido
```



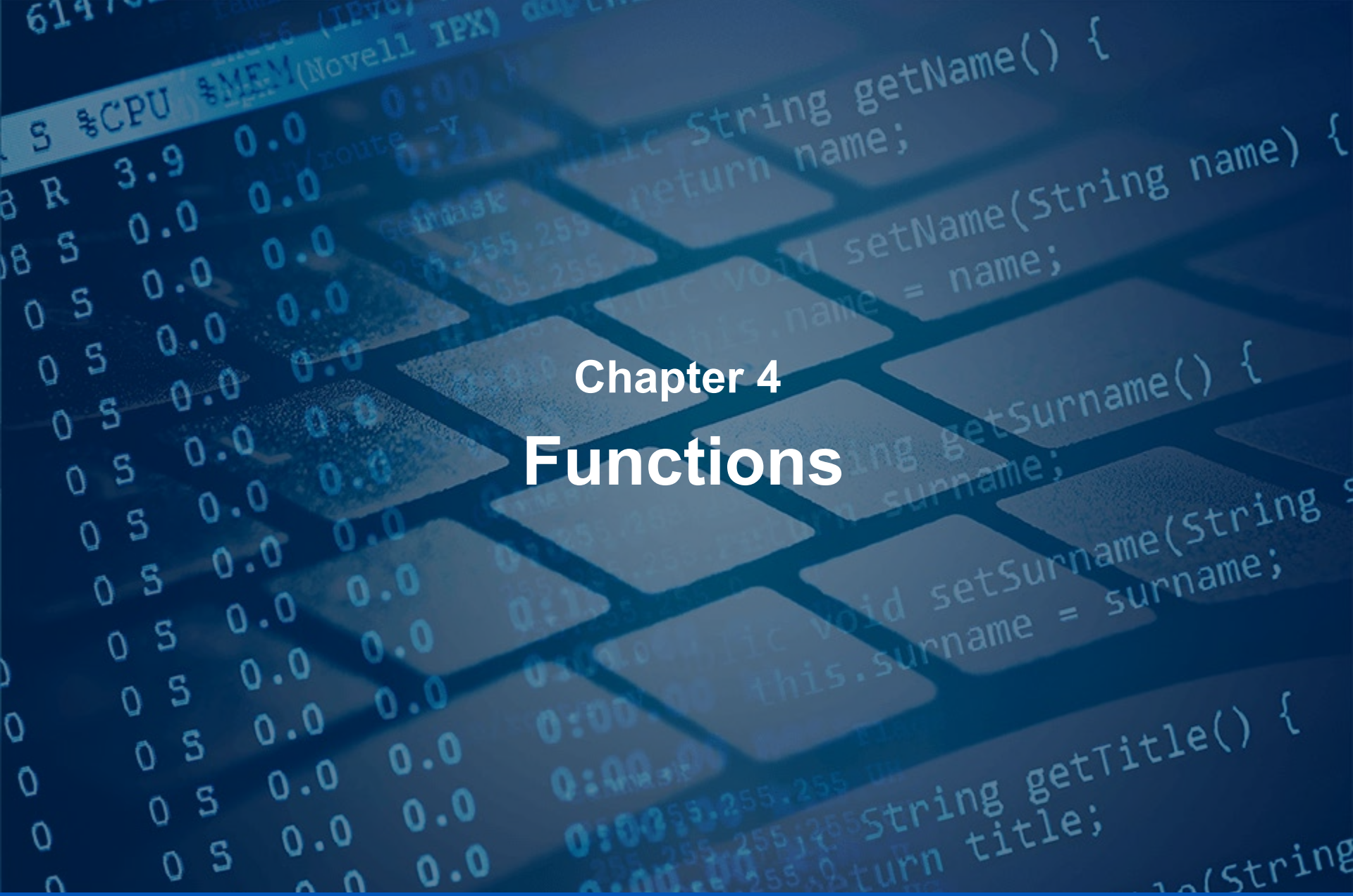
Hands-On Exercise 3.2

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 3.2: Dictionaries, Sets, and Looping

Objectives

- **Create and manage collections**
  - Lists, tuples, sets, and dictionaries
- **Perform iteration**



LEARNING TREE INTERNATIONAL

Objectives

- Create and call a simple function
- Use anonymous lambda functions

Contents

Defining and Calling

- **Lambda Functions**

Function Overview

- **A block of statements that execute as a unit when called and may have**
  - A name
  - An argument list
  - A `return` statement
- **A generic, reusable unit that simplifies program design**
  - Breaks a solution into smaller pieces
  - Hides internal details
  - Provides a single place for modification
- **Created by a `def` statement**
  - Assigns a name to a function
  - Compound statement
  - Indentation defines the function body

Simple Function Example

simple_functions.py

```
def print_one():
    num = 1
    print('the value of num is', num)

def print_two():
    num = 2
    print('the value of num is', num)

print_one()
print_two()
print('The print_one object', print_one)
```

Function is defined and named

Function is called by its name

```
the value of num is 1
the value of num is 2
The print_one object <function print_one at
0x0000017230BCC268>
```

Function is an object at a memory location

Passing Data Into a Function

- Arguments are passed to the function when called
- Function receives arguments from the parameters specified on the `def` statement when the function is executed
- *Positional* parameters are mapped to the argument list based on their position when the function is called

simple_functions.py

```
def printposit(depart, arrive):
    print('depart and arrive by position:', depart, arrive)

printposit('NRT', 'HNL')
```

Positional arguments

```
depart and arrive by position: NRT HNL
```

Keyword Parameters

- **Are mapped to the argument list based on their names**
  - Function call determines keyword or positional style
  - Optional default values may be assigned

simple_functions.py

```
def printkey(depart, arrive):
    print('depart and arrive by keyword:', depart, arrive)

def printdef(depart='LAX', arrive='HNL'):
    print('depart and arrive defaults:', depart, arrive)

printkey(arrive='HNL', depart='NRT')
printdef(depart='AMS')
```

Specify default parameter values

Keyword arguments may be passed in any order

```
depart and arrive by keyword: NRT HNL
depart and arrive defaults: AMS HNL
```

Default is applied for `arrive` within the function

Variable-Length Parameter Lists

- **Functions may be written to accept any number of arguments**
- **Parameter name preceded by * will hold all remaining positional arguments in a tuple**
- **Parameter name preceded by ** will hold all remaining keyword arguments in a dictionary**
- **Function header syntax:**
  - Positional and keyword without defaults must be the leftmost
  - Keywords with defaults follow
  - A single *`parameter` follows
  - A single **`parameter` is rightmost

- **Function call syntax:**
  - Positional arguments must be the leftmost
  - Keyword arguments follow

Variable-Length Parameter List Example

simple_functions.py

```
def printargs(*args, **kwargs):
    print('Positional', args)
    print('Keyword', kwargs)

printargs('Jean', 35, 97.85)
printargs(name='Jean', age=35, rate=97.85)
printargs('Employee', name='Jean', age=35, rate=97.85)

# The following would cause a syntax error
# SyntaxError: positional argument follows keyword argument
# printargs(name='Jean', age=35, rate=97.85, 'Employee')
```

Positional arguments are in a tuple

Keyword arguments are in a dictionary

```
Positional ('Jean', 35, 97.85)
Keyword {}
Positional ()
Keyword {'name': 'Jean', 'age': 35, 'rate': 97.85}
Positional ('Employee',)
Keyword {'name': 'Jean', 'age': 35, 'rate': 97.85}
```

Variable-Length Argument Lists

- Functions may be called with a sequence or dictionary argument
- Argument name preceded by * will pass a collection as a sequence of positional parameters
- Argument name preceded by ** will pass a dictionary as keyword parameters

simple_functions.py

```
employee1 = ['Jean', 35, 97.85]
employee2 = {'name': 'Jules', 'age': 29, 'rate': 89.99}
printargs(*employee1)
printargs(**employee2)
printargs(employee2)
```

employee2 without ** is a single positional

```
Positional ('Jean', 35, 97.85)
Keyword {}
Positional ()
Keyword {'name': 'Jules', 'age': 29, 'rate': 89.99}
Positional ({'name': 'Jules', 'age': 29, 'rate': 89.99},)
Keyword {}
```

Parameters and Scope

- **Parameter is a new reference created for each argument**
  - Created when the function is called
  - Removed when the function completes
- **Parameters are local to the function**

simple_functions.py

increment() scope

```
def increment(number):
    number += 1
    print('function number is', number)
```

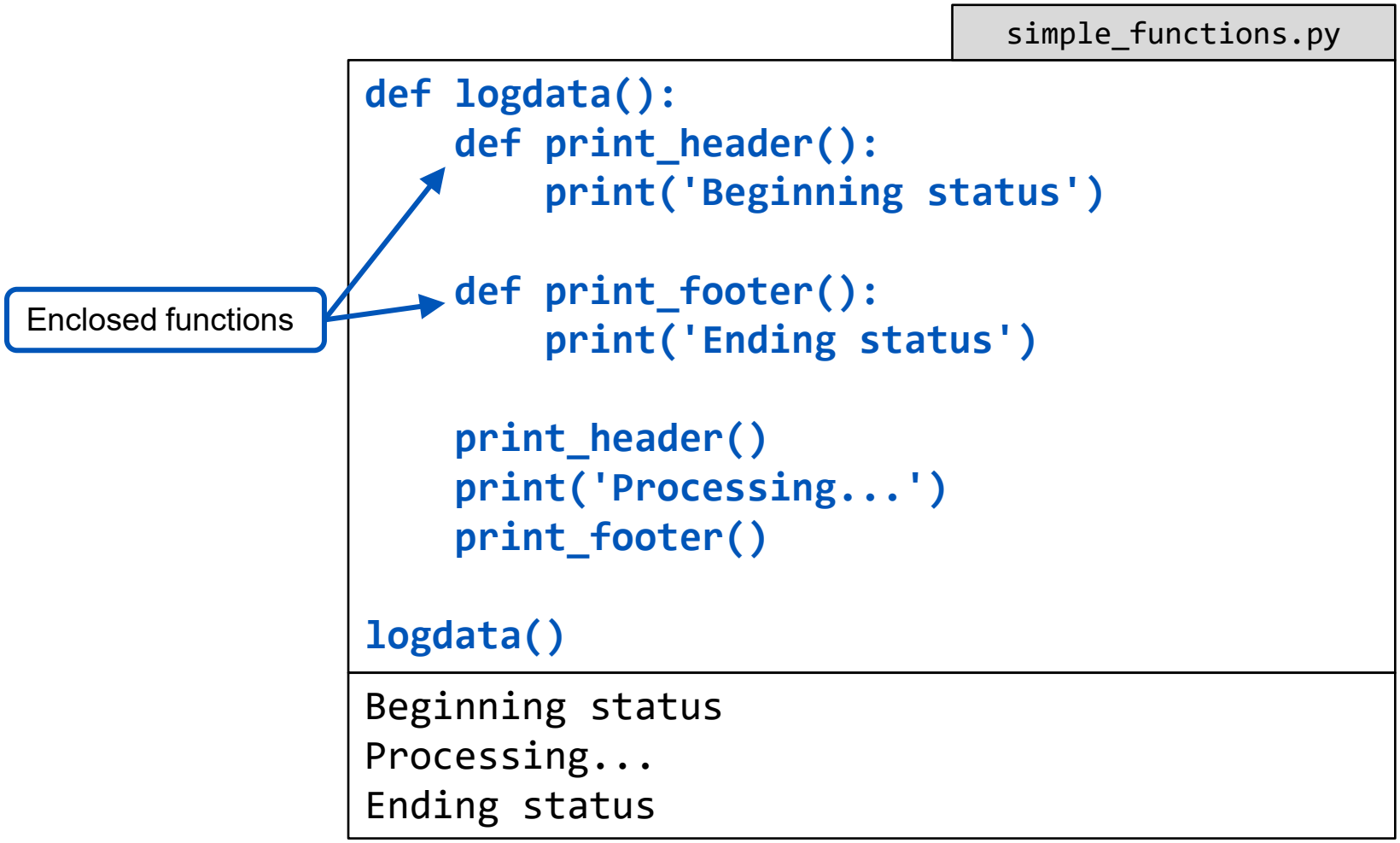
Global scope

```
number = 5
increment(number)
print('global number is', number)
```

```
function number is 6
global number is 5
```

Enclosed Functions

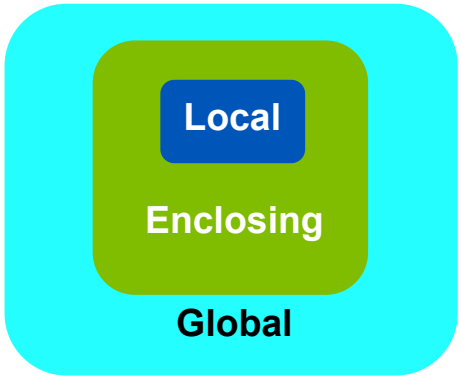
- **A function definition may be within another function**
  - Its name is known in the enclosing function



LEGB Rule

- Describes attribute resolution order

1. Local: within a function
2. Enclosing: within an enclosing function
3. Global: within the module or file
4. Built-in: within the Python `builtin` module



Scope

- **Namespace where an object is known**
- **Based on the location of the assignment**
  - Within an enclosed function
  - Within an enclosing function
  - Outside any function

simple_functions.py

```
3 var = 'global'
def fun1():
 2  var = 'enclosing'

    def fun2():
        var = 'local'
    1   print('enclosed var:', var)

    fun2()
    print('enclosing var:', var)

fun1()
print('global var:', var)
```

Enclosing function

Enclosed function

```
enclosed var: local
enclosing var: enclosing
global var: global
```

`global` Statement

- **Declares a variable as a reference to a global object**
  - For duration of the code block
  - Not thread safe

simple_functions.py

```
var = 'global'
def fun1():
    var = 'enclosing'

    def fun2():
        global var
        var = 'local'
        print(var)

    fun2()
    print(var)

fun1()
print(var)
```

Global object was modified by reference in the function

```
local
enclosing
local
```

nonlocal Statement

- **Declares a variable as a reference in the nearest enclosing scope, excluding global**

simple_functions.py

```
var = 'global'
def fun1():
    var = 'enclosing'

    def fun2():
        nonlocal var
        var = 'local'
        print(var)

    fun2()
    print(var)

fun1()
print(var)
```

References var from the enclosing scope

```
local
local
global
```

`return` Statement

- **Terminates the function execution**
  - Control returns to the point of the call
- **Optionally includes values sent back to the caller**
  - Or `None` if no value is explicitly returned
- **Is optional**
  - Function terminates at the end of the indented block
  - `None` is returned

Function return Example

simple_functions.py

```
def addtwice(num):
    return num + num

def double_vals(arg):
    return arg, arg * 2

ans = addtwice(3)
print('ans is', ans)
first, second = double_vals('a')
print('first is', first, 'second is', second)
```

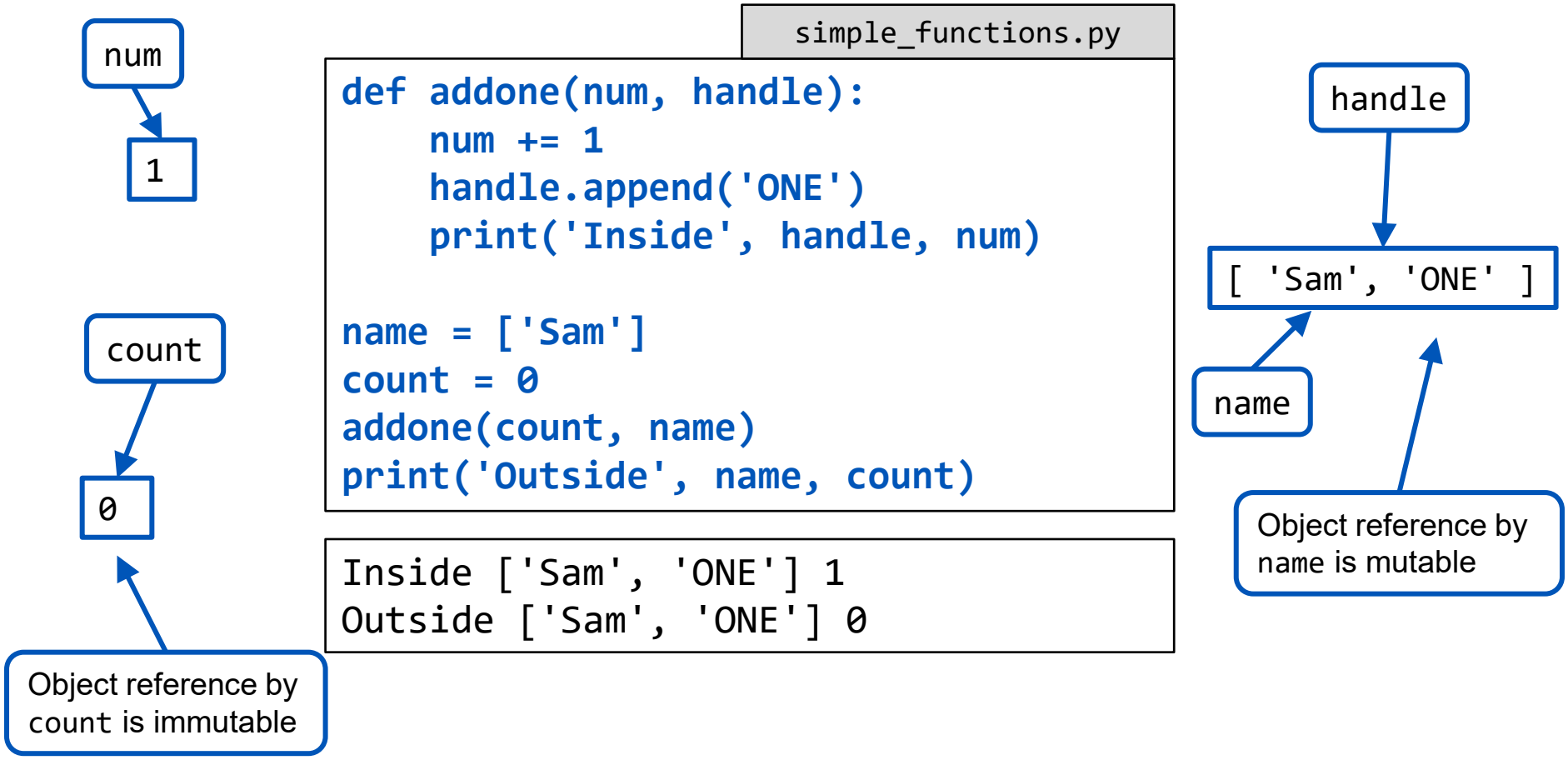
Return a reference to a single object

Return a sequence

```
ans is 6
first is a second is aa
```

Mutable and Immutable Arguments

- **An argument that references a mutable object may have its referenced object changed**



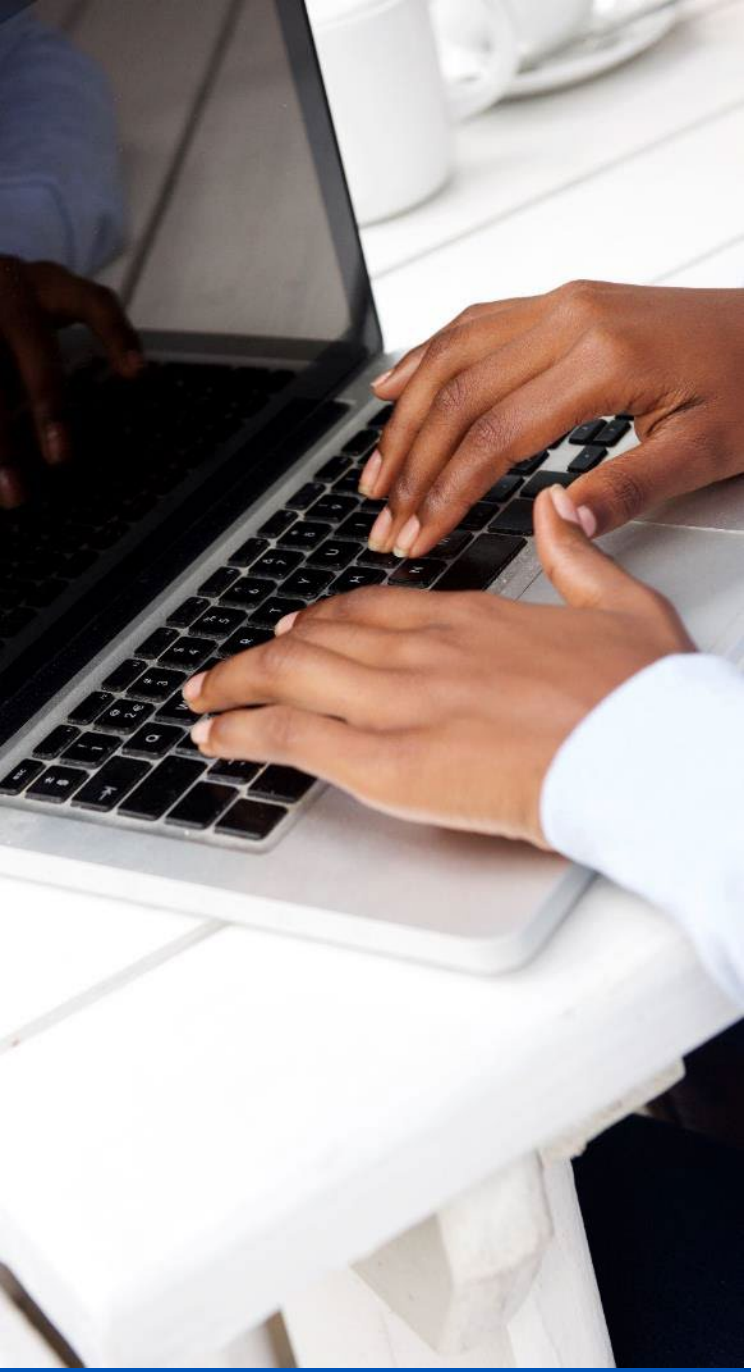
Functions and Polymorphism

- **A single function can work with many types**
  - An example of *polymorphism*
  - Any type of objects may be passed as arguments
  - Any type of object may be returned
- **Only operations within the function are type-dependent**
  - Otherwise, Python raises an exception

simple_functions.py

```
def twice(parm):
    return parm + parm

print('Try twice()', twice(5.5))
print('Try twice()', twice(['a', 'list']))
# The following will cause a Type exception to be raised
# print(twice({'firstname': 'Robert', 'lastname': 'Johnson'}))
```

```
Try twice() 11.0
Try twice() ['a', 'list', 'a', 'list']
```



Hands-On Exercise 4.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 4.1: Creating and Calling Functions

Contents

The Lambda Expression

- **Creates an anonymous function that contains an expression**
  - Reference may be assigned
- **Syntax:**
  - `lambda` *`params`*`:` *`expression`*
- **Used in the same way as regular functions**
  - Arguments may be passed in
  - *`expression`* result is returned
    - `lambda x, y:  x ** 2 + y ** 3`

The sorted() Function

- **sorted(*iterable*, key=*function*, reverse=*Boolean*)**
  - Returns a sorted list
  - Parameter key specifies a function that returns the comparison key
    - lambda may be used

lambda_function_generator.py

```
costs = (('YYZ', '35'), ('HNL', '100'), ('NRT', '52.5'))
print(sorted(costs))
print(sorted(costs, key=lambda p: p[1]))
print(sorted(costs, key=lambda p: float(p[1])))
```

lambda returns comparison key

```
[('HNL', '100'), ('NRT', '52.5'), ('YYZ', '35')]
[('HNL', '100'), ('YYZ', '35'), ('NRT', '52.5')]
[('YYZ', '35'), ('NRT', '52.5'), ('HNL', '100')]
```

[0] is default comparison key

[1] is returned by the lambda

Functions as Arguments

- **A function is an object**
  - Name is a reference to that object
  - Can be used as an argument

lambda_functions.py

```
def print_german():
    print('Guten Morgen')

def print_italian():
    print('Buon Giorno')

def print_greeting(lang, lang_func):
    print('Good Morning in', lang, 'is', end=' ')
    lang_func()

print_greeting('German', print_german)
print_greeting('Italian', print_italian)
```

Reference to the function object used as an argument

Call the function

```
Good Morning in German is Guten Morgen
Good Morning in Italian is Buon Giorno
```

Hiding Function Calls in Lambda Expressions

- ***function*(*args*) can be hidden within a `lambda` expression**
  - Executed when the `lambda` is executed
    - Not when the `lambda` is created

lambda_functions.py

```
def print_german(name):
    print('Guten Morgen', name)

def print_italian(name):
    print('Buon Giorno', name)

def print_greeting(lang, lang_func):
    print('Good Morning in', lang, 'is', end=' ')
    lang_func()

print_greeting('German', lambda: print_german('Hans'))
print_greeting('Italian', lambda: print_italian('Gina'))
```

Reference to the `lambda` object is parameter

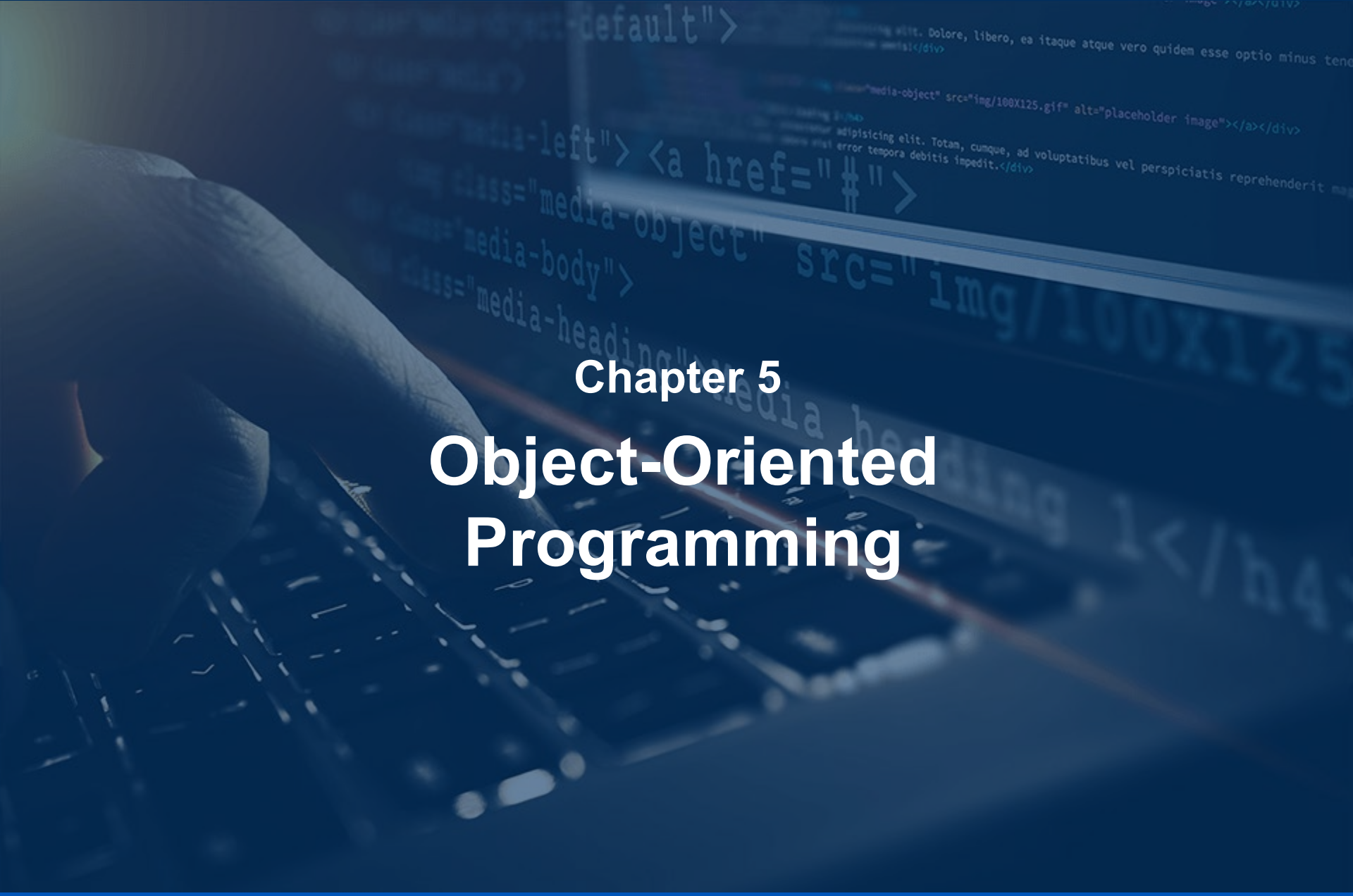
Function call with argument is the `lambda` body

Executes the `lambda`

```
Good Morning in German is Guten Morgen Hans
Good Morning in Italian is Buon Giorno Gina
```

Objectives

- Create and call a simple function
- Use anonymous lambda functions



Chapter 5

Object-Oriented Programming

LEARNING TREE INTERNATIONAL

Objectives

- Define a class
- Create subclasses through inheritance
- Attach methods to classes

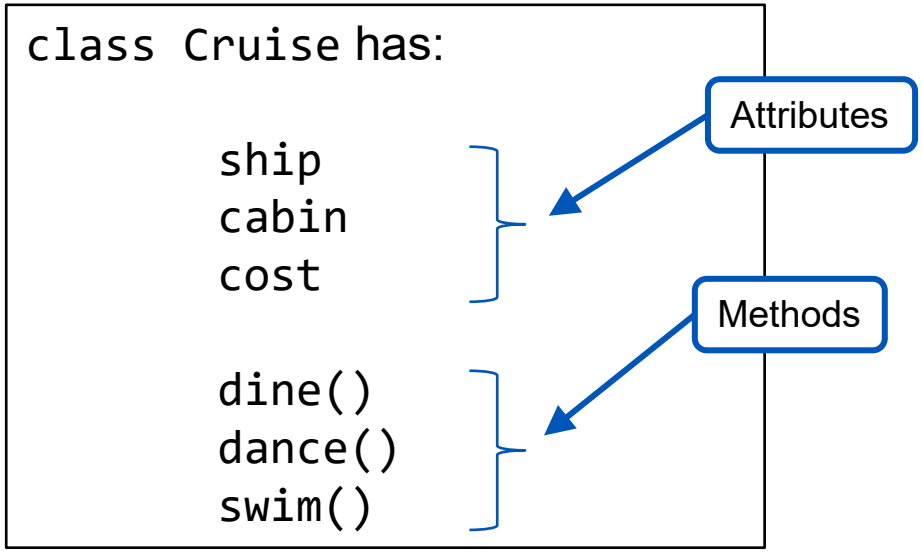
Contents

Classes and Instances

- Inheritance

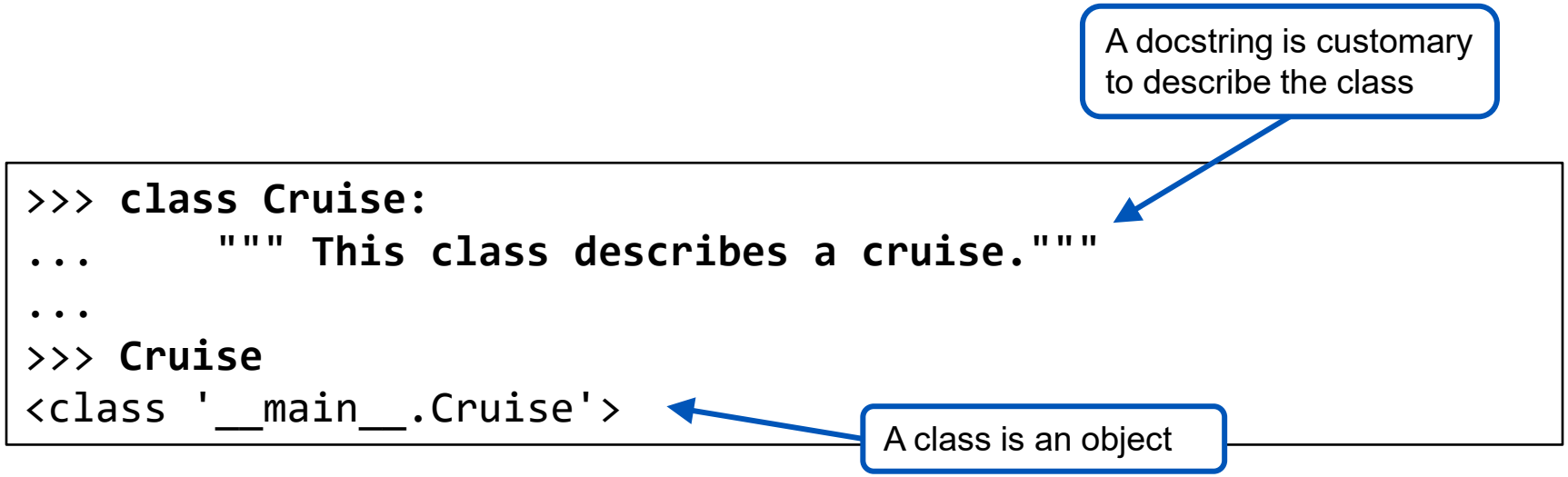
Class

- **A generic description of an object—a *type***
  - A template for objects
- **A container with**
  - Attributes that describe the object's state
  - Methods that describe the object's behavior
- ***Has A* relationship**
- **Main building block of an Object-Oriented Programming (OOP) solution**



The `class` Statement

- **Creates a new object template**
- **Assigns a name to the class**
  - PEP 8 recommends *CapWords* style



The `__init__()` Method

- **Called automatically when an instance is created**
  - A *constructor*
  - Python calls *`class`*`.__init__(`*`instance, args`*`)`
- **Used to assign initial attribute values**
  - Based on its argument list
    - Defaults may be provided

simple_classes.py

```
class Cruise:
    """ This class describes a cruise."""
    def __init__(self, ship=None, cost=0.0, cabin=0):
        self.ship = ship
        self.cost = cost
        self.cabin = cabin
```

Keyword parameters with defaults

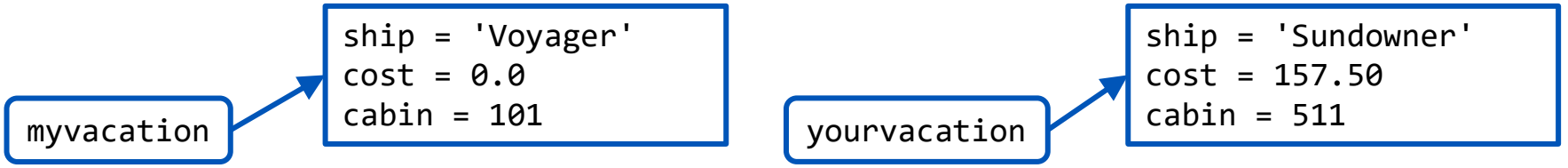
The `self` Parameter

- **References the particular instance making the call**
  - First parameter of an instance method

simple_classes.py

```
myvacation = Cruise(ship='Voyager', cabin=101)
yourvacation = Cruise(ship='Sundowner',
                      cost=157.50, cabin=511)
print(myvacation.ship, myvacation.cabin, myvacation.cost)
print(yourvacation.ship, yourvacation.cabin, yourvacation.cost)
```

```
Voyager 101 0.0
Sundowner 511 157.5
```



__init__() Parameter Styles

- Keyword or positional arguments may be used

simple_classes.py

```
class Cruise:
    """ This class describes a cruise."""
    def __init__(self, shipname, price, room):
        self.ship = shipname
        self.cost = price
        self.cabin = room

myvacation = Cruise(shipname='Voyager', price=0, room=101)
yourvacation = Cruise('Sundowner', 157.50, 511)
print(myvacation.ship, myvacation.cabin, myvacation.cost)
print(yourvacation.ship, yourvacation.cabin, yourvacation.cost)
```

Attribute names

```
Voyager 101 0.0
Sundowner 511 157.5
```

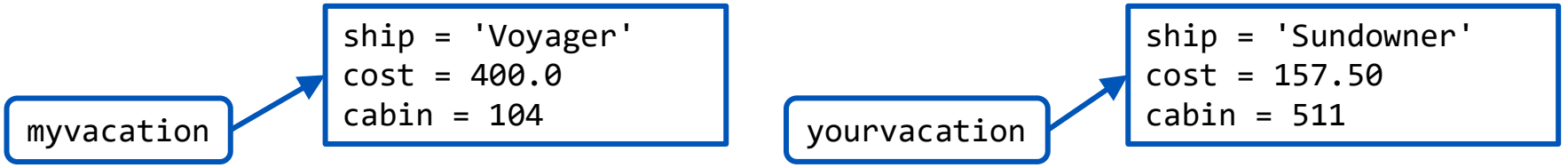
Modifying Instance Attributes

- **Assigned into the instance namespace**
  - Affects only that instance

simple_classes.py

```
myvacation.cost = 400.0
myvacation.cabin = 104
print(myvacation.ship, myvacation.cabin, myvacation.cost)
print(yourvacation.ship, yourvacation.cabin, yourvacation.cost)
```

```
Voyager 104 400.0
Sundowner 511 157.5
```



Methods

- **Functions bound to a class**
  - Created by a `def` statement within the `class` statement
  - Are available for any instance
    - `self` parameter references the instance

simple_classes.py

```
class Cruise:
    """ This class describes a cruise."""
    def __init__(self, ship=None, cost=0.0, cabin=0):
        self.ship = ship
        self.cost = cost
        self.cabin = cabin

    def dine(self, amount):
        self.cost += amount
```

Modifies the instance attribute

Methods Illustrated

- **Called as *instance.method(args)***
  - Python passes *instance* as the first argument
    - `Cruise.dine(myvacation, 125.0)`

simple_classes.py

```
myvacation = Cruise(ship='Voyager', cabin=101)
yourvacation = Cruise(ship='Sundowner',
                      cost=157.50, cabin=511)
myvacation.dine(125.0)
yourvacation.dine(215.50)
print('myvacation', myvacation.cost)
print('yourvacation', yourvacation.cost)
```

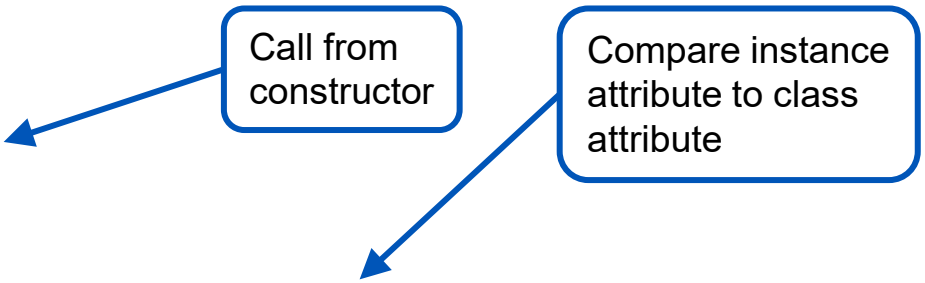
```
myvacation 125.0
yourvacation 373.0
```

Class Attributes

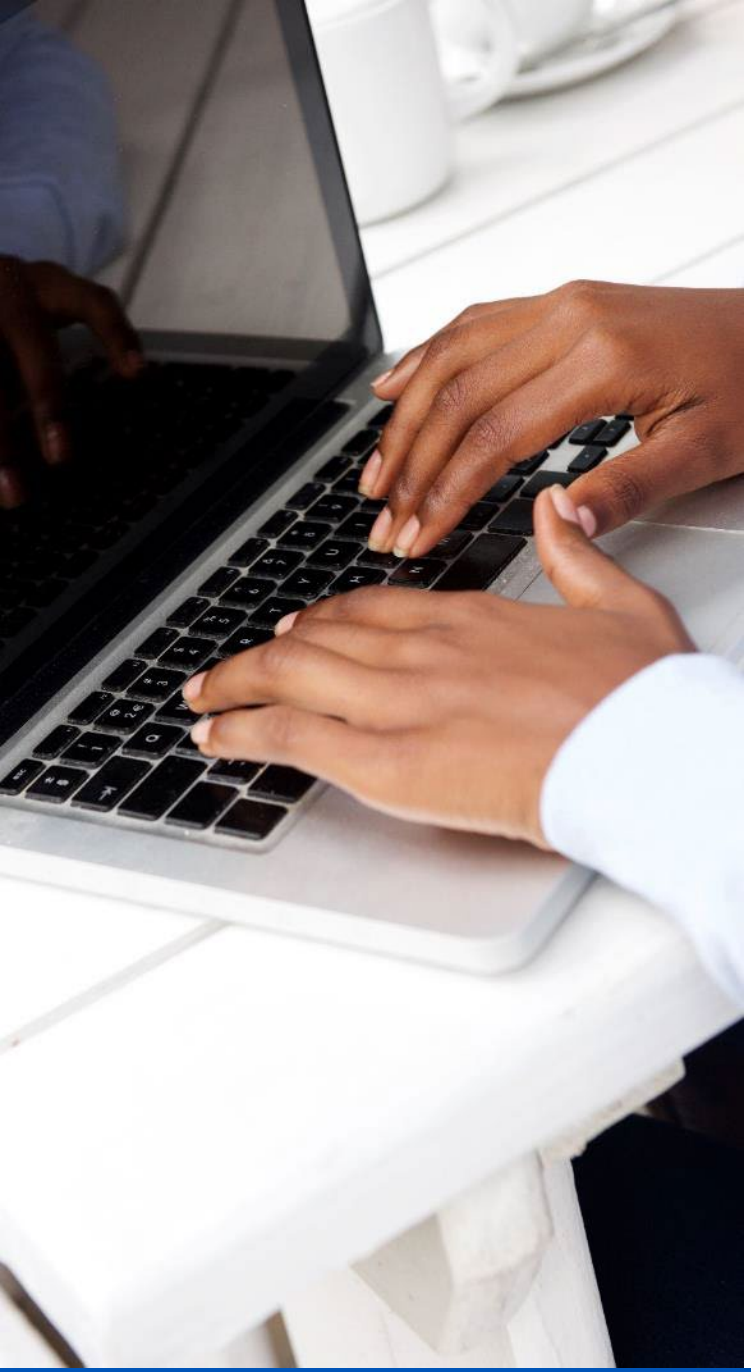
- **Attribute encapsulated within a class**
  - Instances share the reference
  - Accessed through class name qualification

simple_classes.py

```
class Cruise:

    premiumcabins = (101, 102, 105, 106, 109, 110)

    def __init__(self, ship=None, cost=0.0, cabin=0):
        self.ship = ship
        self.cost = cost
        self.cabin = cabin
        self.charge_upgrade()

    def charge_upgrade(self):
        if self.cabin in Cruise.premiumcabins:
            self.cost += 50.0

myvacation = Cruise(ship='Voyager', cabin=101)
print(myvacation.cost)
```




```
50.0
```



Hands-On Exercise 5.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 5.1: Classes and Initialization

Contents

- Classes and Instances

Inheritance

Class Hierarchy

- **Describe the *Is A* relationship**
  - Derived/subclass is an extension of the parent/base class
    - Adds additional attributes and methods
    - Subclass performs class/type specific operations
- **Syntax:**

```
class BaseClass:
    ...
class SubClass(BaseClass):
```

Inherit from parent class

```
class Trip:
    ...
class Flight(Trip):
    ... additional attributes ...
class Cruise(Trip):
    ... additional attributes ...
class SunsetSail(Cruise):
    ... additional attributes ...
```

Class Inheritance

- Methods and attributes from the parent class are available in subclasses

simple_inheritance.py

```
class Trip:
    def __init__(self, departday=None, arriveday=None):
        self.departday = departday
        self.arriveday = arriveday
    def print_departure(self):
        print('Trip leaves on', self.departday)

class Cruise(Trip):
    def print_schedule(self):
        print('Cruise', self.departday, 'to', self.arriveday)

class Flight(Trip):
    def print_arrival(self):
        print('Flight arrives on', self.arriveday)
```

Inheritance Hierarchy

- **Subclasses *without* `__init__()` call the inherited `__init__()` from the parent class**

simple_inheritance.py

```
voyage = Cruise(departday='Friday', arriveday='Monday')
voyage.print_departure()
voyage.print_schedule()

flthome = Flight(departday='Monday', arriveday='Monday')
flthome.print_departure()
flthome.print_arrival()
```

Method inherited from `Trip`

Method within `Flight`

```
Trip leaves on Friday
Cruise Friday to Monday
Trip leaves on Monday
Flight arrives on Monday
```

Subclass Instance Initialization

inheritance_without_super.py

```
class Trip:
    def __init__(self, departday=None, arriveday=None):
        self.departday = departday
        self.arriveday = arriveday

    def print_departure(self):
        print('Trip leaves on', self.departday)

class Cruise(Trip):
    def __init__(self, departday, arriveday, ship=None):
        self.ship = ship
        Trip.__init__(self, departday=departday,
                      arriveday=arriveday)

    def print_schedule(self):
        print('Cruise', self.departday, 'to', self.arriveday)
```

Assign `ship` attribute only

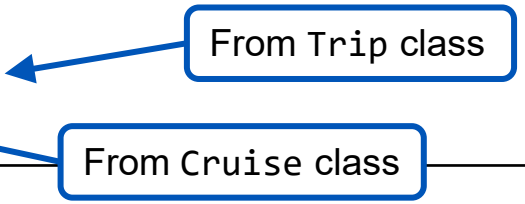
Explicitly call parent class constructor with `self` as first argument

Subclass Extension

- **Subclasses may add additional attributes**
- **Subclasses *with* `__init__()` may call the parent class `__init__()`**
  - Not called automatically

inheritance_without_super.py

```
voyage = Cruise(departday='Friday', arriveday='Monday',
                ship='Sea Breeze')

print(voyage.departday)
print(voyage.ship)
```




```
Friday
Sea Breeze
```

The super() Function

- **Returns an object that delegates methods to a parent class**
  - Without explicitly naming the parent class

```
class Parent:
    def __init__(self, ...

class Subclass(Parent):
    def __init__(self, ...
        super().__init__( ...
```

Calls the constructor from its parent class

No need to pass `self` as the first argument

Subclass Instance Initialization Using super()

inheritance_with_super.py

```
class Trip:
    def __init__(self, departday=None, arriveday=None):
        self.departday = departday
        self.arriveday = arriveday

    def print_departure(self):
        print('Trip leaves on', self.departday)

class Cruise(Trip):
    def __init__(self, departday, arriveday, ship=None):
        self.ship = ship
        super().__init__(departday=departday,
                         arriveday=arriveday)

    def print_schedule(self):
        print('Cruise', self.departday, 'to', self.arriveday
```

Parameters for `Cruise` and `Trip`

Pass additional arguments to parent class constructor

Subclass Instance Initialization Using super()

inheritance_with_super.py

```
voyage = Cruise(departday='Friday', arriveday='Monday',
                ship='Sea Breeze')
voyage.print_departure()
voyage.print_schedule()
```

```
Trip leaves on Friday
Cruise Friday to Monday
```

Calling Superclass Methods Using *args and **kwargs

inheritance_with_super.py

```
class Trip:
    def __init__(self, departday=None, arriveday=None):
        self.departday = departday
        self.arriveday = arriveday

    def print_departure(self):
        print('Trip leaves on', self.departday)

class Cruise(Trip):
    def __init__(self, ship=None, *args, **kwargs):
        self.ship = ship
        super().__init__(*args, **kwargs)

    def print_schedule(self):
        print('Cruise', self.departday, 'to', self.arriveday
```

Defaults in parent class

Reference remaining arguments

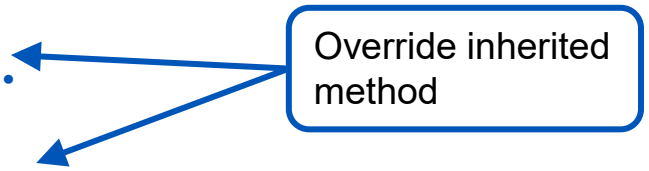
Pass remaining arguments to parent class constructor

Overriding Methods

- Single operation name may replace the same named operation from a parent class
- Attribute lookup order determines which is found first

override.py

```
class Trip:
    def __init__(self, ...

    def print_trip(self ...

class Cruise(Trip):
    def __init__(self, ...

    def print_trip(self):
        print('Ship is', self.ship)

day1 = Cruise(departday='Friday', arriveday='Saturday',
                  ship='Moonbeam')
day1.print_trip()
```



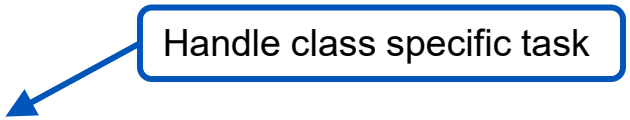

```
Ship is Moonbeam
```

Overriding Methods

- **Methods in subclass perform type-specific operations**
  - Parent class provides common operations
  - `super()` may be used to access the parent's methods

override.py

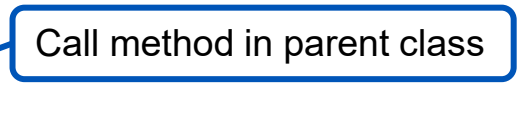
```
class Trip:
    def __init__(self, departday=None, arriveday=None):
        self.departday = departday
        self.arriveday = arriveday

    def print_trip(self):
        print('Schedule is', self.departday, self.arriveday,
              end=' ')
```

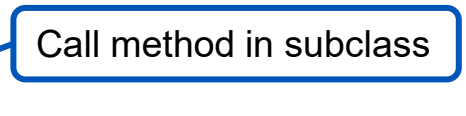


Overriding Methods

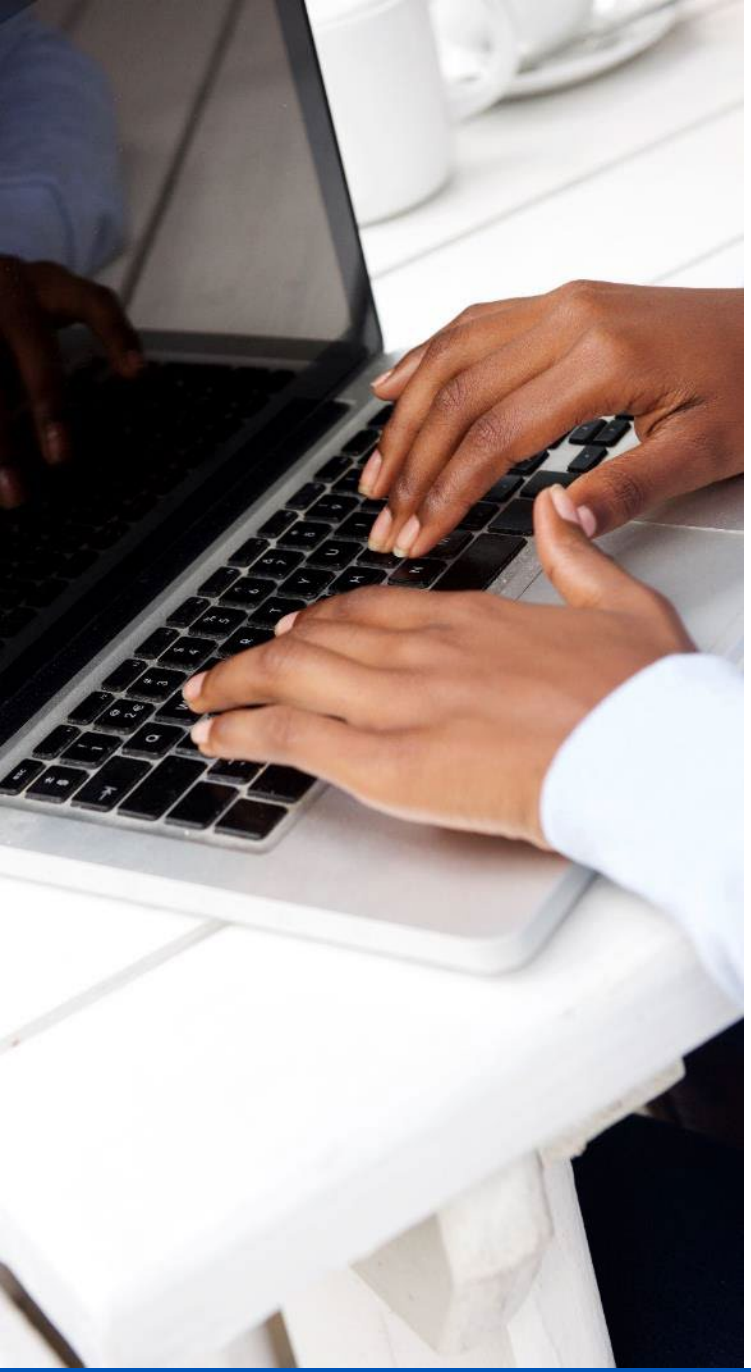
override.py

```
class Cruise(Trip):
    def __init__(self, ship=None, *args, **kwargs):
        self.ship = ship
        super().__init__(*args, **kwargs)

    def print_trip(self):
        super().print_trip()
        print('Ship is', self.ship)

travels = [Cruise(departday='Friday', arriveday='Saturday',
           ship='Moonbeam'),
           Cruise(departday='Wednesday', arriveday='Friday',
           ship='Golden Sun')]

for travel in travels:
    travel.print_trip()
```






```
Schedule is Friday Saturday Ship is Moonbeam
Schedule is Wednesday Friday Ship is Golden Sun
```



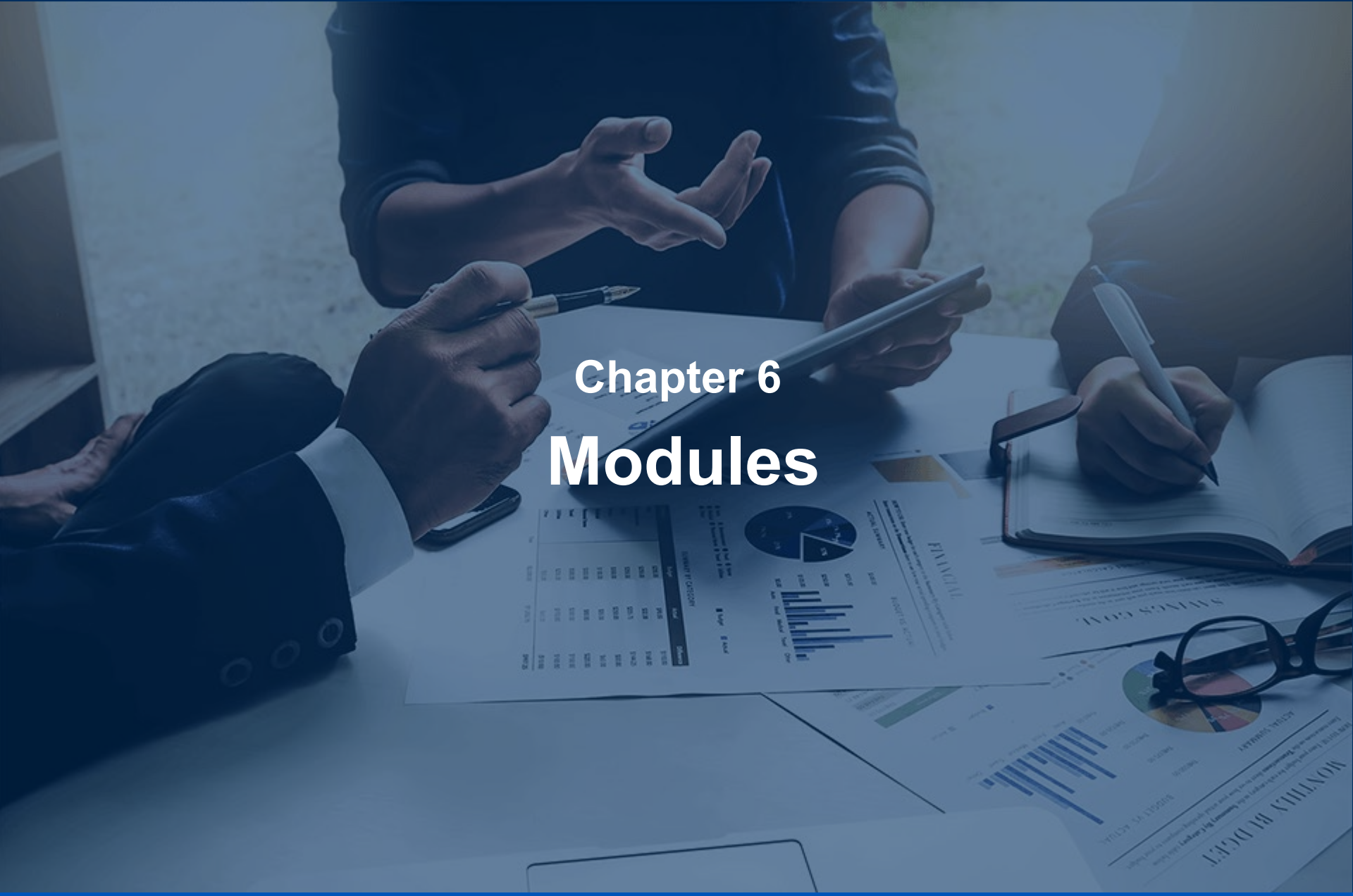
Hands-On Exercise 5.2

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 5.2: Inheritance

Objectives

- Define a class
- Create subclasses through inheritance
- Attach methods to classes



Chapter 6
Modules

LEARNING TREE INTERNATIONAL

Objectives

- Create new modules
- Access additional modules
- Use the standard library

Contents

Module Overview

- `import` and Namespace
- The Standard Library and Installing Additional Packages

Module

- **Highest-level programming unit**
  - Modules have classes and functions
  - Functions have statements
  - Statements have expressions
- **Library providing a set of services**
  - Included functions provide each service
- **Single container of reusable code**
  - One place to manage changes
  - May be shared with other modules
    - Reduces repetition

Module Files

- **Could be**
  - Source code file, *mod*`.py`, or byte code file, *mod*`.pyc`
  - Dynamically linked library, *mod*`.dll` or *mod*`.so`
    - Extension modules
- **Located by a Python search in**
  - The current directory
  - One of the directories contained in `PYTHONPATH`
  - The standard library

Contents

The `import` Statement

- **Syntax: `import` *`modulename`***
- **Creates an object of the module's contents**
- **Enables access to the *`modulename`*'s classes and functions**
- **Possibly creates the `.pyc` byte code file**
  - If the corresponding `.py` file is newer, recompile the `.pyc`
- **Executed once per process**
  - Later `imports` of same module name use the existing object

Module Execution

- **All unenclosed statements are executed**
- **Attributes are created**
  - `functions` or `objects`
- **Occurs in a separate namespace**
- **Creates a module object**
  - Based on the file name

circles.py

```
PI = 3.14

def area(radius):
    return PI * radius ** 2

def diameter(radius):
    return radius * 2
```

Attributes

importing1.py

```
import circles

print(circles)
```

Module object created

```
<module 'circles' from  … \\Chap6_Examples\\circles.py'>
```

Module Attributes

- Reside within the module namespace
- Are accessed through a qualified name
  - *module.attribute*

importing1.py

```
print(circles.PI)
rad = 1.0
ar = circles.area(rad)
print('circle is radius {} area {}'.format(rad, ar))
```

```
3.14
circle is radius 1.0 area 3.14
```

Multiple imports

- Qualified attributes reference the proper module

circles.py

```
PI = 3.14

def area(radius):
    return PI * radius ** 2

def diameter(radius):
    return radius * 2
```

volumes.py

```
import circles

def cylinder_volume(radius, length):
    return circles.area(radius) * length

def sphere_volume(radius):
    return circles.area(radius) * radius * 4 / 3
```

Multiple imports

- Qualified attributes reference the proper module

importing2.py

```
import circles
import volumes

rad = 1.0
length = 2.0
ar = circles.area(rad)
print('circle area {}'.format(ar))
vol = volumes.cylinder_volume(rad, length)
print('cylinder volume {}'.format(vol))
```

```
circle area 3.14
cylinder volume 6.28
```

Chained `imports`

- **Imported file contains an `import` statement**
  - A imports B; B imports C
- **Require two levels of qualification to get embedded attributes**
  - From A, use *`B.C.attribute`*

importing3.py

```
import volumes

rad = 1.0
length = 2.0
ar = volumes.circles.area(rad)
print('circle area {}'.format(ar))
vol = volumes.cylinder_volume(rad, length)
print('cylinder volume {}'.format(vol))
```

The `import as` Statement

- **Syntax: `import` *`modulename`* `as` *`name`***
  - Access contents using *`name`* instead of *`modulename`*

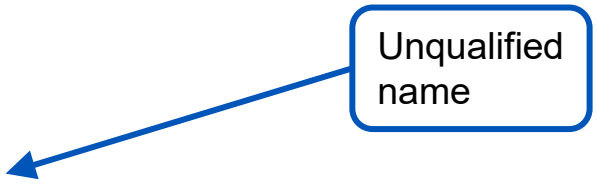
```
import volumes as vo

ar = vo.circles.area(rad)

vol = vo.cylinder_volume(rad, length)
```

`from` Statement

- **Copies named attribute into the current namespace**
  - No attribute qualification needed
- **Syntax: `from` *`module`* `import` *`attributes`***

importing4.py

```
from circles import area
from volumes import cylinder_volume as cv

rad = 1.0
length = 2.0
ar = area(rad)
print('circle area {}'.format(ar))
vol = cv(rad, length)
print('cylinder volume {}'.format(vol))
```




```
circle area 3.14
cylinder volume 6.28
```

from Statement

- **from *module* import ***
  - Imports all attributes into the current namespace
  - Not a best practice
    - May corrupt the namespace

logger.py

```
import sys
def log(msg):
    print(msg, file=sys.stderr)
```

importing5.py

```
from logger import *
log('test 1 error message')

from math import *
log('test 2 error message')
```

Called log() from math

```
...
log('test 2 error message')
TypeError: must be real number, not str
```

Exception raised

Examining Namespace

- **The `dir()` function displays names of module attributes**
  - The `__dict__` attribute is a dictionary of a module's objects

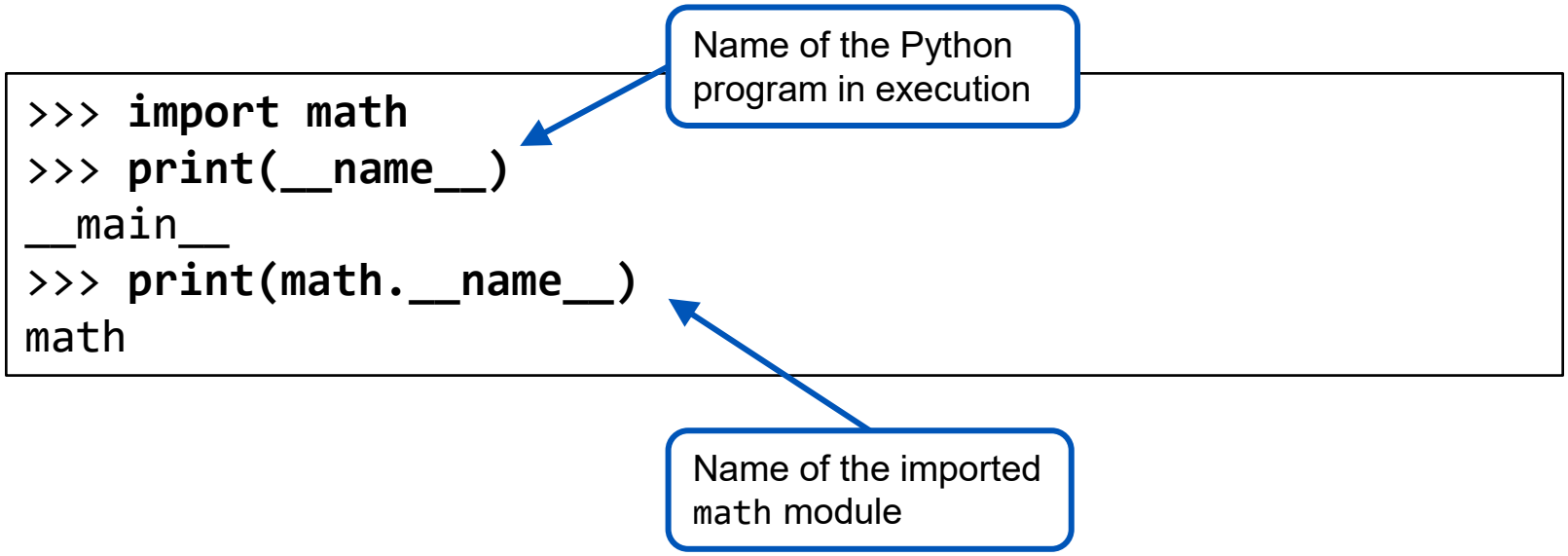
```
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'math']
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh' .......
...
>>> for key, value in math.__dict__.items():
...     print(key, '\t ===>', value)
...
pow      ===> <built-in function pow>
fsum     ===> <built-in function fsum>
cosh     ===> <built-in function cosh> ...
```

Examine the current module

Examine the `math` module

The __name__ Attribute

- **__name__ is equal to '__main__' when a module is running as a standalone program**
  - It is set to the module name when the module was imported



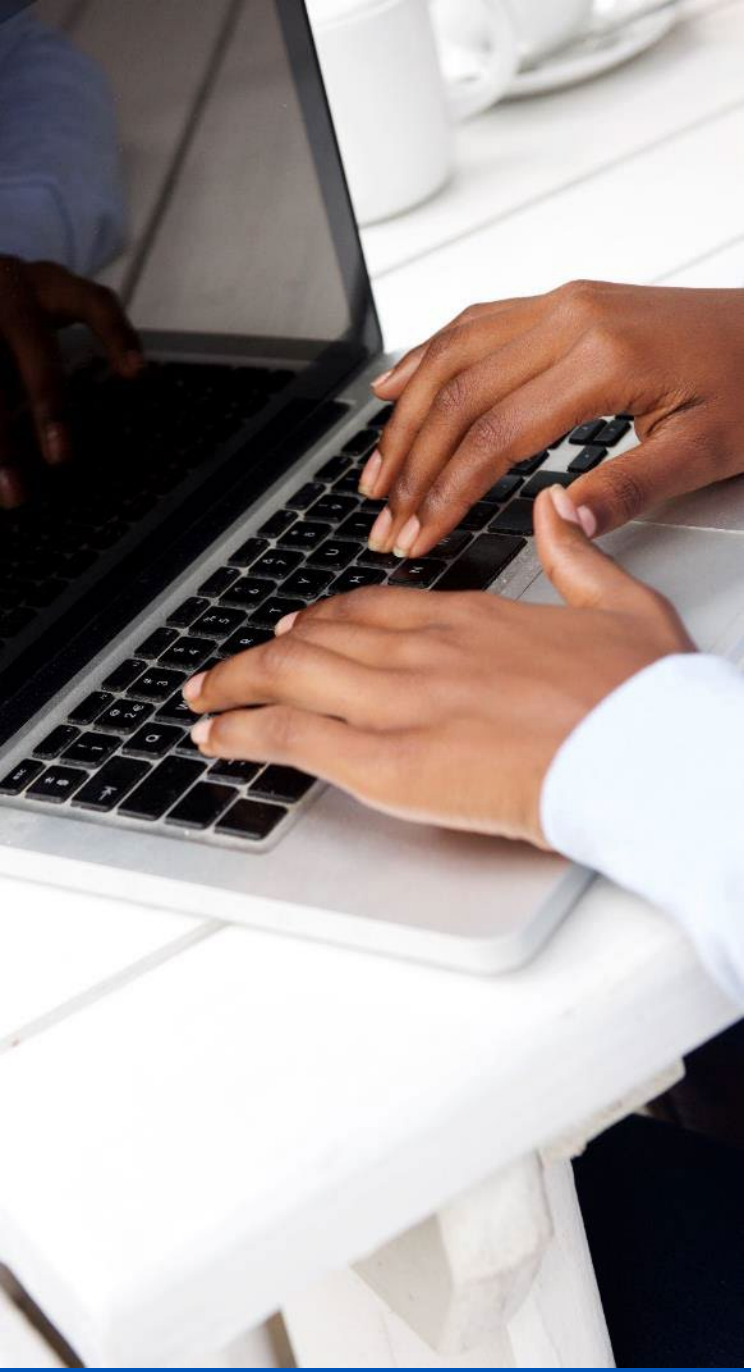
Testing the __name__ Attribute

- **May be tested to conditionally execute module testing code**
  - Only when not imported

name_check.py

```
class Person:
    def __init__(self, name):
        self.name = name

def check_person():
    testname = 'Katrina'
    student = Person(testname)
    if student.name == testname:
        print('Person constructor ok')

if __name__ == '__main__':
    check_person()
```

Not executed if this module was imported

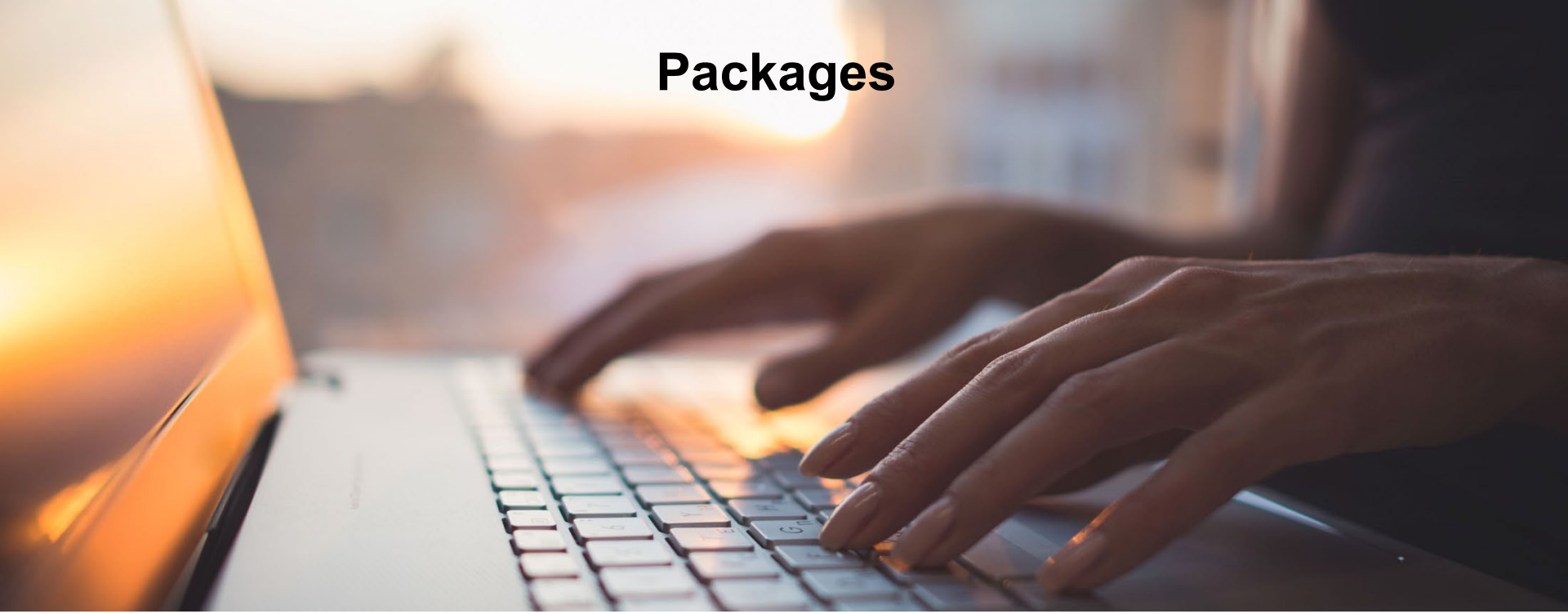


Hands-On Exercise 6.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 6.1: Modules

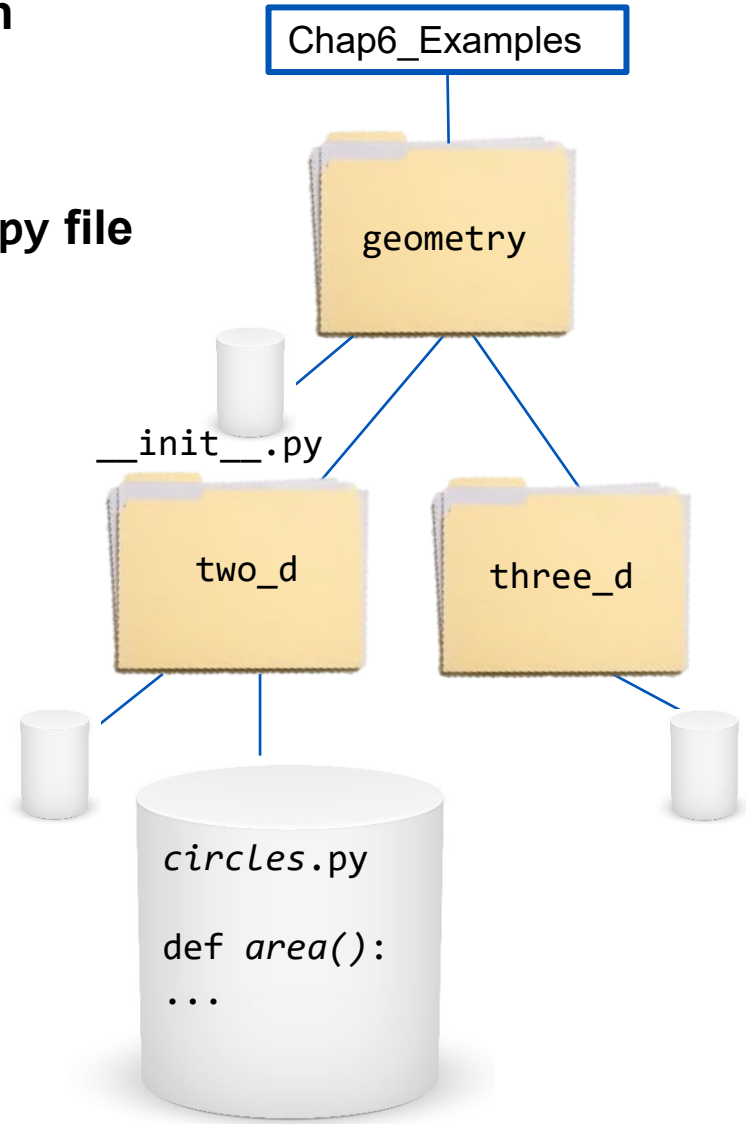
Packages



- **Directory hierarchy of module files**
- **Allow module hierarchy to follow the directory structure**
  - Instead of flat structure of modules
  - Group related modules as needed
    - System architecture, service, Python version, etc.

Package Directories

- **Package directory must be within the search path**
  - `site-packages` is common
- **Package directory must have the `__init__.py` file**
  - May be an empty file
- **Runs the code in all `__init__.py` files within the path**
  - Creates an object of the module's contents
- **Each directory becomes a namespace**



Package import

- **`import` *`pkg`***
  - Each `.` separates a level of directory structure

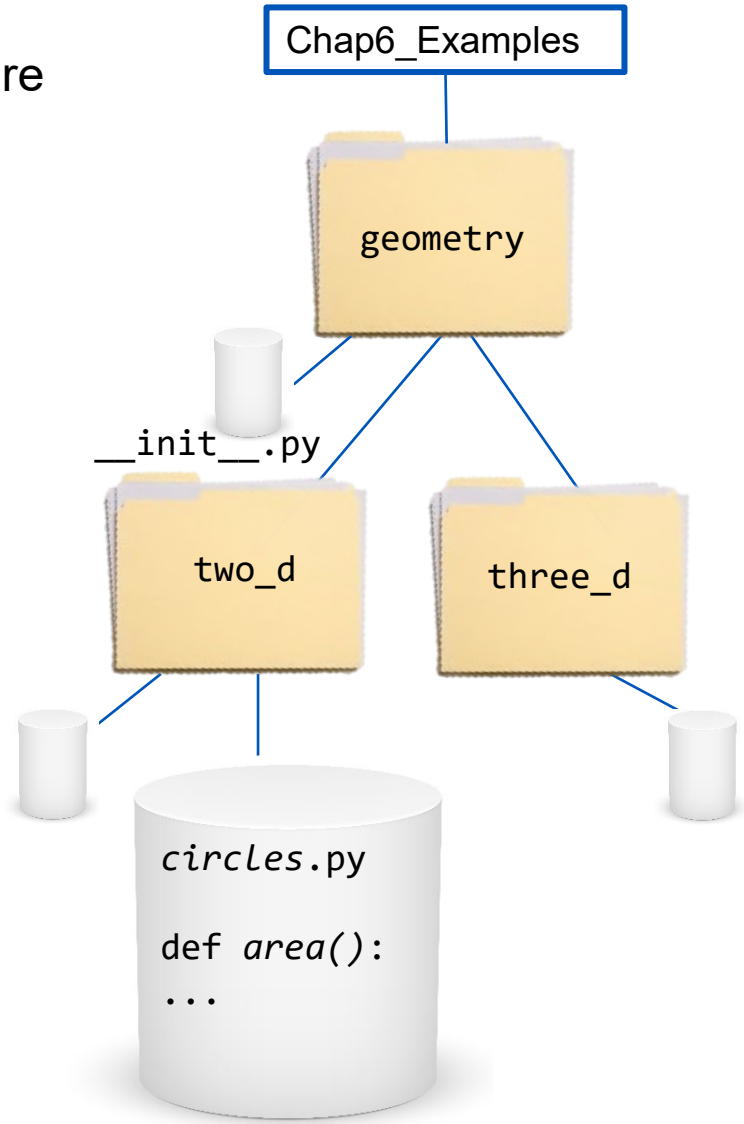
```
import geometry
geometry.two_d.circles.area()
```

- **`import` *`pkg.dir.module`***

```
import geometry.two_d.circles as ci
ci.area()
```

- **`from` *`pkg.dir.module`* *`import`* `...`**

```
from geometry.two_d.circles import area
area()
```



Package Example

- **Chap6_Examples folder contains the geometry package**

pkg_importing.py

```
import geometry.two_d.circles as ci
from geometry.three_d.volumes import cylinder_volume as cv

rad = 1.0
length = 2.0
ar = ci.area(rad)
print('circle area {}'.format(ar))
vol = cv(rad, length)
print('cylinder volume {}'.format(vol))
```

```
circle area 3.14
cylinder volume 6.28
```

Contents

- Module Overview
- `import` and Namespace

The Standard Library and Installing Additional Packages

Standard Library

- **Collection of modules that come with Python**
- **Not part of the core language itself**
- **Interfaces to access common utilities**
  - Operating system
  - Database access
  - Date and time information and measurement
  - GUI and network application development
  - Regular expression pattern matching
  - Archiving and compression
  - And more!
- **Details in the standard documentation**

Using Standard Library and Documentation

Do Now

- Checking for a file's existence
- The `os.path.isfile()` function returns a Boolean to indicate if its argument is an existing file

1. Find the Python documentation on the `os.path` module
2. Within the section, find the documentation on the `isfile()` function in that module
3. Open the Python console in PyCharm
4. Import the needed module and call the function to test whether this file exists:

   `C:\Course\1905\Files\Chap6_Examples\importing1.py`

   HINT: If using `\` in the pathname, be sure the string is a raw string. It turns out, `isfile()` will also accept `/` in the pathname. Try it both ways.

Using Standard Library and Documentation—Bonus—1

Do Now

- **Create all the various groups of three toppings that can be served on a pizza**

1. **Find the Python documentation on the `itertools` module**
2. **Find the documentation on the `combinations()` function in that module**
3. **Open the `Chap6_Examples\pizza.py` file for editing in PyCharm**
   - Notice the `toppings` list has been assigned
4. **Add the coding to display all the combinations of three toppings without repeating any combination**

Using Standard Library and Documentation—Bonus—2

Do Now

► **Convert strings into `datetime` objects and check which date is later**

1. **Find the Python documentation on the `datetime` module**
2. **Find the documentation on the `datetime.fromisoformat()` method in that module**
3. **Open the `Chap6_Examples\dt_convert.py` file for editing in PyCharm**
   - Notice the string assignments
4. **Add the coding to:**
   - Convert the two strings
   - Compare them to see which is an earlier date and time

Downloading Packages

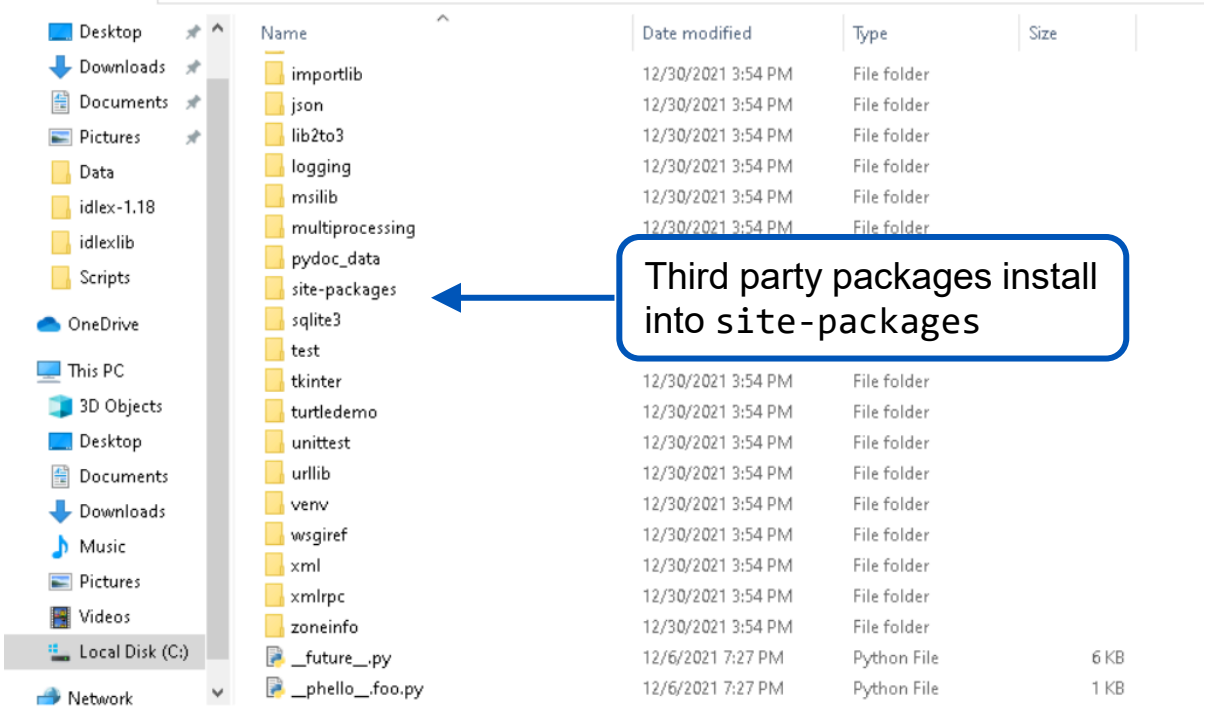
- **Python Package Index is a package repository with tutorials**
  - For downloading and installing
  - For creating and uploading
  - https://pypi.org
- **`pip` is a fundamental tool for package management**
  - Installs packages and dependencies from the repository
  - Removes packages

```
pip install packagename
```

Standard Library Directory Structure

- **Examine the package directory structure**

```
C:\Python\Python312\Lib\site-packages
```



Objectives

- Create new modules
- Access additional modules
- Use the standard library

Chapter 7

Managing Exceptions and Files

Objectives

- Handle and raise exceptions
- Perform file I/O

I/O = input/output

Contents

Exceptions

- Files

Exceptions

- **Are errors generated at runtime**
- **May be raised by Python itself or manually from within a program**
- **Cause a change in the control flow of a program**
  - Default action is immediate termination
    - Includes a stack trace of the calls leading to the exception
- **It is the programmer's responsibility to provide code to handle exceptions**
- **Python's exception-handling capabilities**
  - Simplify coding
  - Increase robustness
  - Provide a uniform approach to handling errors across application code

Exception Example

exceptions1.py

```
def print_age_in_days(years):
    print('Age in days is ', 365.25 * int(years))

age = input('Enter your age: ')
print_age_in_days(age)
```

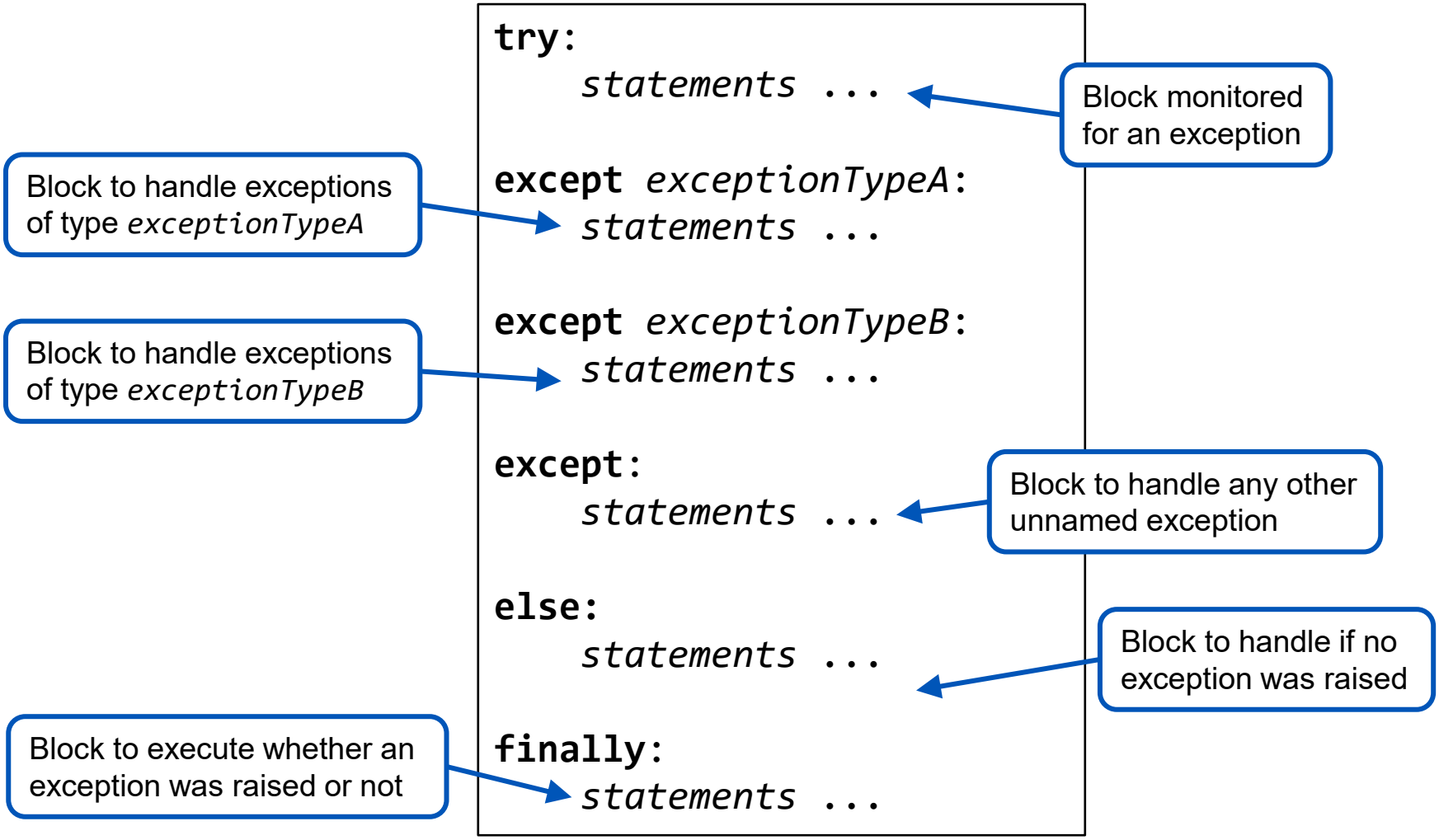
Line that caused the exception

```
Enter your age: thirty
Traceback (most recent call last):
  File "C:/Course/1905/Files/Chap7_Examples/exceptions1.py",
...
line 2, in print_age_in_days
    print('Age in days is ', 365.25 * int(years))
ValueError: invalid literal for int() with base 10: 'thirty'
```

Exception type

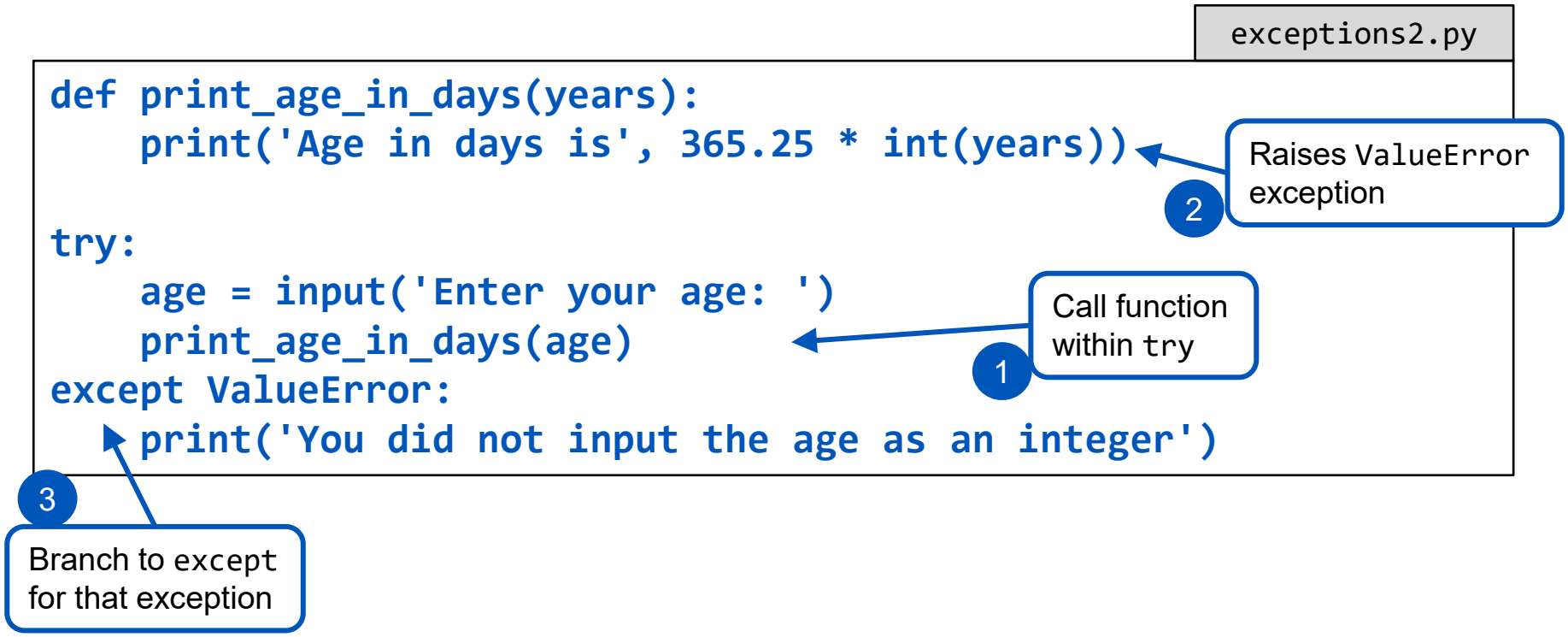
The `try` Statement

- **General structure of exception-handling code is as follows:**



Handling a Single Exception

- **Statements within the `try` block are executed and monitored for an exception**
- **On exception, control passes to the appropriate `except` block**
  - Associated with the most enclosing `try`



Handling Multiple Exception Types

- If present, multiple `except` blocks are checked sequentially
- `except(`*`exceptionA, exceptionB`*`)` defines a single block for multiple exceptions
- `except:` defines a block for any unnamed exception

exceptions3.py

```
def print_age_in_days(years):
    print('Age in days is', 365.25 * int(years))

try:
    age = input('Enter your age: ')
    print_age_in_days(age)
except ValueError:
    print('You did not input the age as an integer')
except EOFError:
    print('End of file from standard input')
except:
    print('Non Value or EOF error occurred')
```

For any unnamed exception type

The `else` and `finally` Clauses

- **`else`: defines a block that is executed if no exceptions are raised**
  - Follows all `except` clauses
    - Must have at least one `except`
- **`finally`: defines a block that is always executed**
  - Whether an exception was raised or not

exceptions4.py

```
def print_age_in_days(years):
    print('Age in days is', 365.25 * int(years))

try:
    age = input('Enter your age: ')
    print_age_in_days(age)
except ValueError:
    print('You did not input the age as an integer')
else:
    print(age, 'was successfully converted to integer')
finally:
    print('Input test complete')
```

Block always executes

No exception raised

Exception Instances

- **Exception instances are assigned by `except` *`ExceptionType`* `as` *`name`***
  - *`name`*`.args` references a tuple given to the *`ExceptionType`* constructor

exceptions5.py

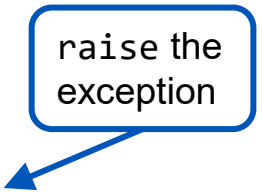
```
def print_age_in_days(years):
    print('Age in days is', 365.25 * int(years))

try:
    age = input('Enter your age: ')
    print_age_in_days(age)
except ValueError as ve:
    print('You did not input the age as an integer')
    print('Value Error handled', ve.args)
else:
    print(age, 'was successfully converted to integer')
finally:
    print('Input test complete')
```

The raise Statement

- **Initiates the named exception**
  - Which may be handled or not

exceptions6.py

```
import string
def print_age_in_days(years):
    for digit in years:
        if digit not in string.digits:
            raise ValueError('Cannot convert', digit)
    print('Your age in days is more than', 365 * int(years))

try:
    age = input('Enter your age: ')
    print_age_in_days(age)
except ValueError as ve:
    print('You did not input the age as an integer')
    print('Value Error handled', ve.args)
```




```
Enter your age: thirty
You did not input the age as an integer
Value Error handled ('Cannot convert', 't')
```

Exception Class Hierarchy

- **Exceptions are all classes**
  - The ones we handle are all subclasses of the base class `Exception`
- **Can create your own `Exception` subclasses**
  - For exceptions not within Python
  - You must `raise` and handle them explicitly
- **Using a parent class with `except` will also handle exceptions of any subclass**
  - `LookupError` has two subclasses
    - `IndexError`—raised by an invalid index on a sequence
    - `KeyError`—raised by an invalid key on a dictionary

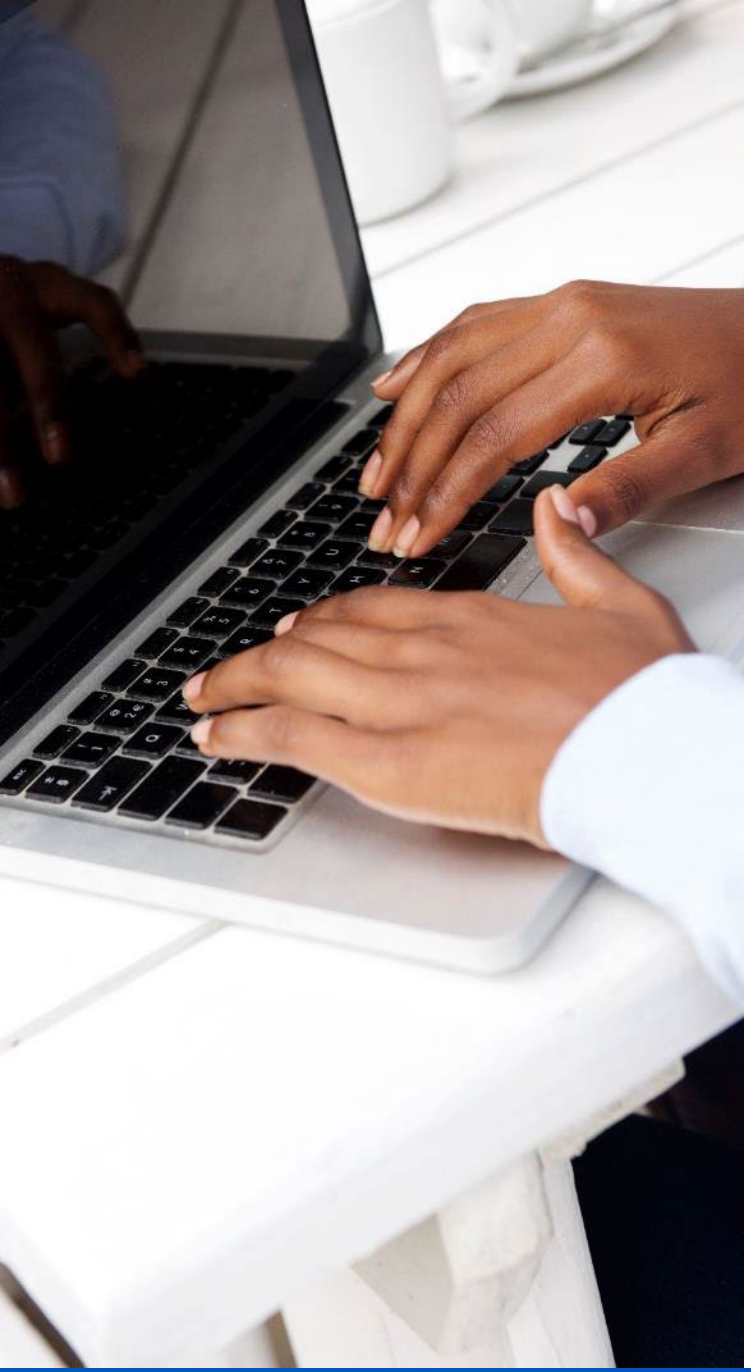
Exception Class Hierarchy

exceptions7.py

```
# Exception Class Hierarchy
def convert_any_digits(indx, digits):
    return int(digits[indx])

try:
    print(convert_any_digits(1, ['12', '34']))  # Works
    print(convert_any_digits('b', {'a': '12', 'b': '34'})) # Works
    print(convert_any_digits(3, ['12', '34'])) # IndexError
  # print(convert_any_digits('c', {'a': '12', 'b': '34'})) # KeyError
except LookupError as le:
    print('Look Up Error handled', le.args)
except ValueError as ve:
    print('Value Error handled', ve.args)
```

ValueError or LookUpError possible

```
34
34
Look Up Error handled (list index out of range',)
```



Hands-On Exercise 7.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 7.1: Exceptions

Contents

- Exceptions

Files

Files

- **Built-in object type**
  - Has methods to handle reading, writing, and positioning within the file
- **Reference contents of many types**
  - Character
  - Numeric
  - Class object

The open and close Statements

- **Open a file syntax: `object = open('pathname' [, 'mode'])`**
- **Returns a file**
- **Specifies an opening *mode***
  - `'r'`—Opened for reading at the beginning
    - Default mode
  - `'w'`—Opened for writing, truncating the file first
  - `'a'`—Opened for writing at the end
  - Additional `'+'` with mode opens for both reading and writing operations
    - `'w+'`—Reading and writing at the beginning, with truncation
    - `'r+'`—Reading and writing at the beginning, without truncation
- **Specifies a file's content type within the *mode***
  - Text is the default
  - `'b'` specifies binary
- **Close a file syntax: `object.close()`**
  - Releases open file reference

Opening Files and Exceptions

- **An `IOError` exception is raised when opening to**
  - Read a file that does not exist
  - Read or write a file without appropriate access rights
- **`IOError` exception attributes include**
  - `errno`—Error message number, `args[0]`
  - `strerror`—Error message string, `args[1]`
  - `filename`—Filename used when the exception was raised

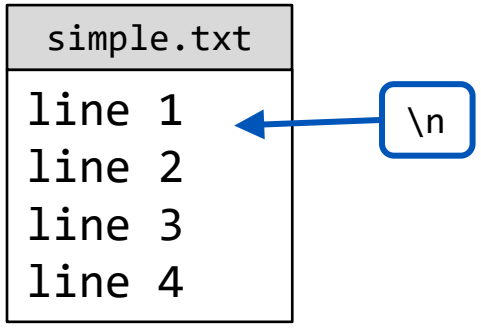
files1.py

```
try:
    infile = open('Incorrectfilename')
except IOError as ioe:
    print('Unable to open the file')
    print('Error number', ioe.args[0])
    print('Message', ioe.args[1])
    print('Filename in error', ioe.filename)
```

If `open()` failed, `close()` is not needed

Reading a Text File

- **File is a sequence of characters**
  - `'\n'` separates lines



l	i	n	e		1	\n	l	i	n	e		2	\n	...

- **`read()`: Returns the entire file contents as a single string**
- **`readline()`: Returns the next line from the file**
  - Includes the `'\n'` line delimiter
  - `rstrip()` string method can remove the `'\n'`
- **`readlines()`: Returns the entire file contents as a list of strings, including the `'\n'`**
- **`IOError` exception may be raised**

Reading a Text File Example

files2.py

```
try:
    infile = open('C:/Course/1905/Data/simple.txt', 'r')
    print(infile.readline().rstrip())
    print(infile.readlines())
    infile.close()
except IOError as ioe:
    print('Error number', ioe.args[0])
    print('Message', ioe.args[1])
```

```
line 1
['line 2\n', 'line 3\n', 'line 4\n']
```

Writing a Text File

- **`write(`*`string_ref`*`)`: Writes a single string into a file**
- **`writelines(`*`list_ref`*`)`: Writes a list's contents into a file**
- **On writing, data is buffered**
  - `close()` writes the buffer and releases the file object
  - `flush()` followed by `os.fsync()` writes the buffer and keeps the file open
- **`IOError` exception may be raised**

Data Handling Exceptions

- Once opened, files should be closed

files3.py

```
try:
    infile = open('C:/Course/1905/Data/simple.txt', 'r')
    try:
        print(infile.readline().rstrip())
        print(infile.readlines())
        infile.write('line 5\n')
    except IOError:
        print('Read or Write error on file')
    finally:
        infile.close()
except IOError as ioe:
    print('Failed to open the file', ioe.args)
```

`IOError` exception raised writing to the file

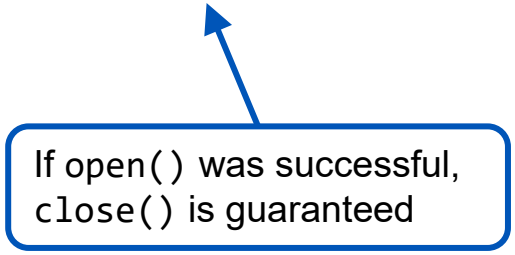
File must be closed whether or not exception occurs

Using `with` to Open and Close Files

- **The `with` statement wraps a block of statements with methods defined by a *context manager***
  - If the file is opened, it will be closed
    - Even if an exception is raised

files4.py

```
try:
    with open('C:/Course/1905/Data/simple.txt', 'r') as infile:
        print(infile.readline().rstrip())
        print(infile.readlines())
        infile.write('line 5\n')

except IOError as ioe:
    print('Read or Write error on file', ioe.args)
```



Using Loops and Iterators for File Access

- The file object is iterable

files5.py

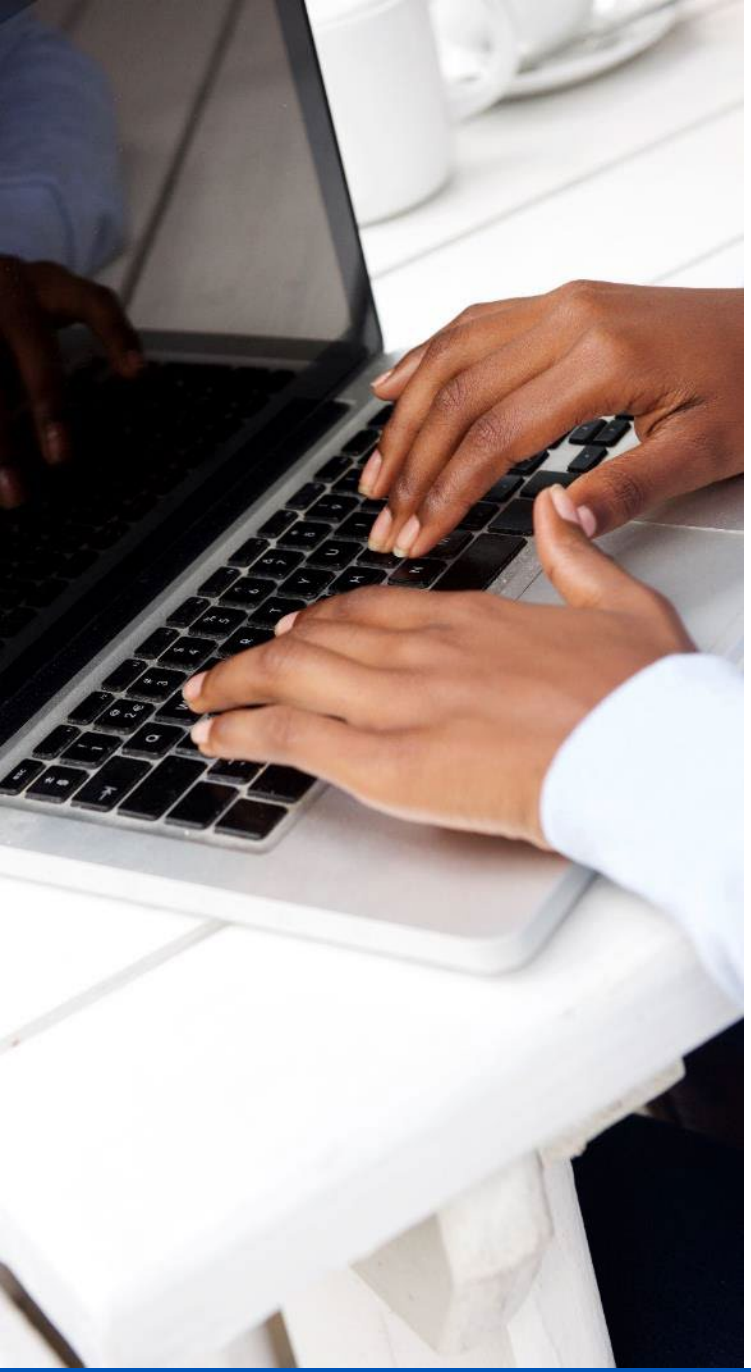
```
try:
    with open('C:/Course/1905/Data/simple.txt', 'r') as infile:
        for dataline in infile:
            print(dataline.rstrip())

except IOError as ioe:
    print('Read or Write error on file', ioe.args)
```

```
line 1
line 2
line 3
line 4
```

The Standard Streams

- **Standard streams are file objects available from the `sys` module**
  - Already opened for reading or writing when the program starts
  - Treated as text files
- **Default to the keyboard and screen when using the Python interpreter**

1. **`sys.stdin`**
   - Provides standard input file for file methods
2. **`sys.stdout`**
   - Provides standard output file for file methods
   - Used by `print`
3. **`sys.stderr`**
   - Provides standard error file for file methods
   - Used for exception messages



Hands-On Exercise 7.2

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 7.2: Managing Files

Objectives

- Handle and raise exceptions
- Perform file I/O

I/O = input/output

Chapter 8

Accessing Relational Databases With Python

LEARNING TREE INTERNATIONAL

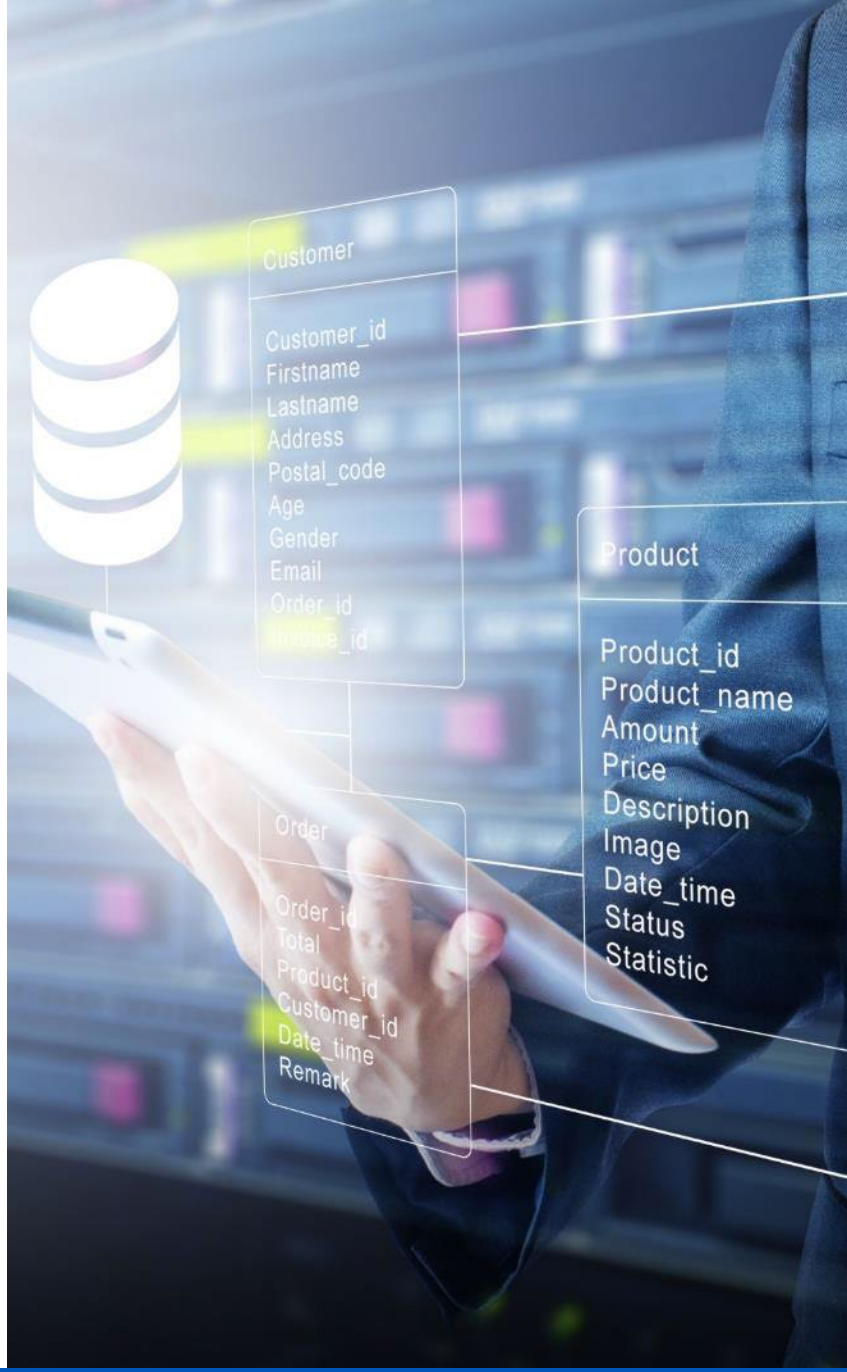
Objectives

- Access databases with Python using SQL

Contents

Relational Databases

- **Using SQL**



Relational Databases

- **Store data as tables**
  - Rows are indexed by keys
    - Unique fields of data
- **Access by the structured query language (SQL)**
  - Standard programming language
- **Are implemented by many proprietary as well as open-source products**
  - Oracle, Sybase, and SQL Server are well-known commercial products
  - PostgreSQL and MySQL are well-known open-source products
  - Modules are needed to access a particular database

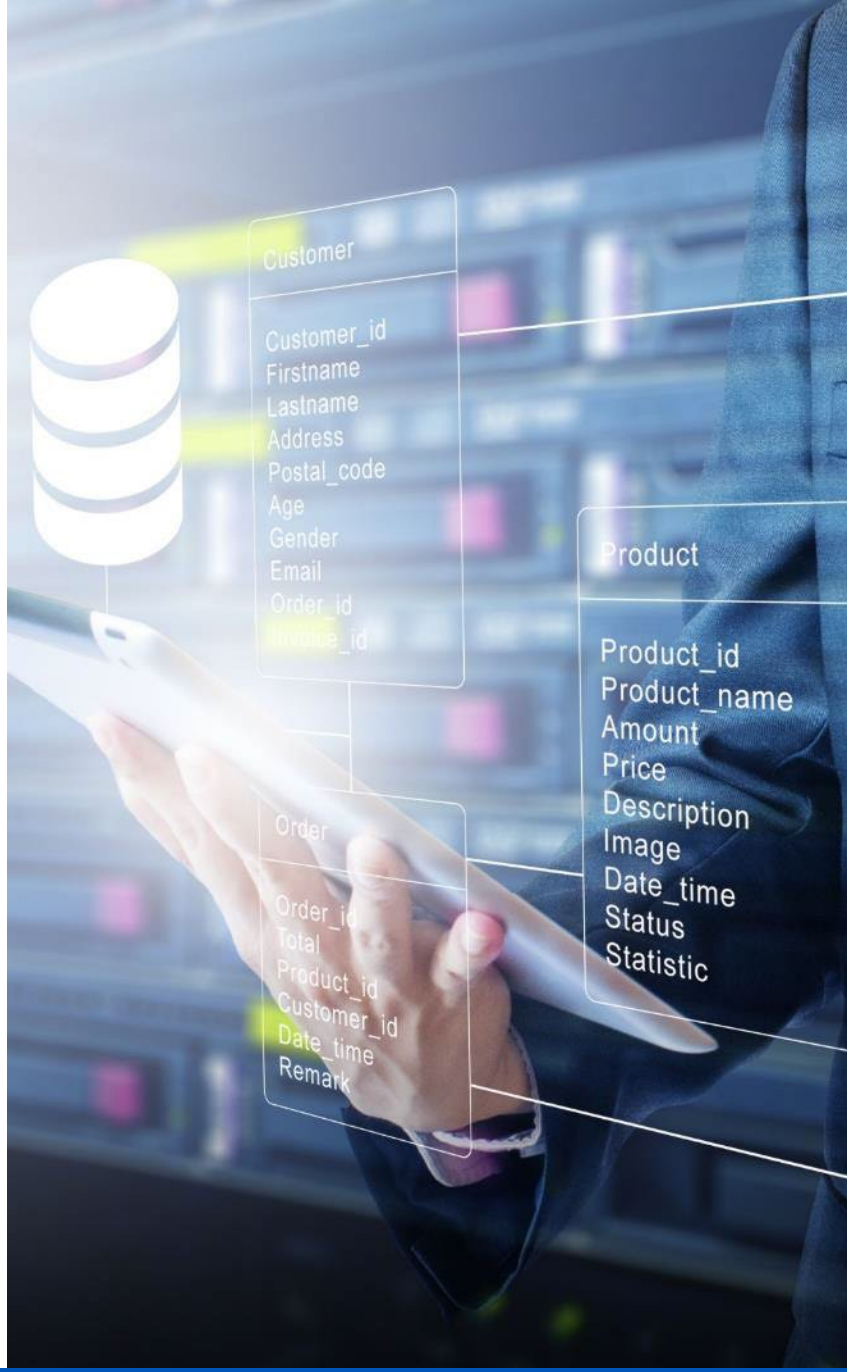
SQLite3

- **Part of Python's standard library**
- **Provides a lightweight database that doesn't require a separate server process**
- **Can be used as a data store**
- **Can be used to prototype an application**
  - Then port the code to a larger database such as PostgreSQL or Oracle
- **`import sqlite3`**

Contents

- Relational Databases

Using SQL



Steps to Accessing the Database

1. Establish a connection
2. Create a cursor for the data interchange
3. Use SQL to access the data
4. Close the connection

Step 1: Establish a Connection

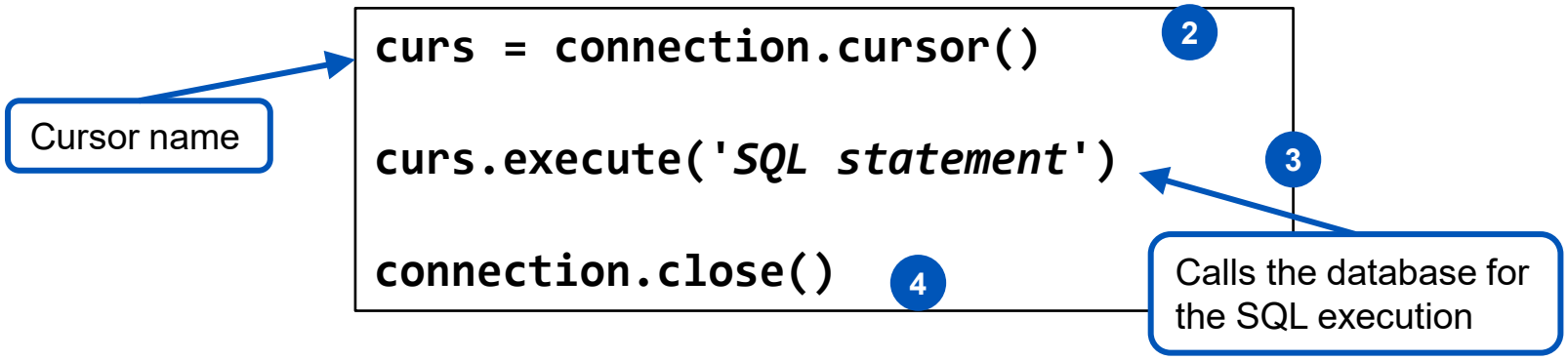
- **`connect()` initiates contact with the database**
  - Requires database name and login information
    - Format is database dependent
  - Returns a connection object
    - Or raises an `OperationalError` exception
- **Connection provides methods for data access management**
  - `close()`—terminates the connection
  - `commit()`—forces write to database store
  - `rollback()`—removes changes back to last `commit()`

```
import database_module

connection = database_module.connect(database_name)
```

1

Steps 2, 3, and 4

- **`cursor()` method creates cursor object**
- **Controlling structure for database access**
- **Provides `execute()` method for SQL statements**
  - `ProgrammingError` is raised for invalid statement



Extracting Data From the Cursor

- **Database rows matching the `SELECT` criteria are available through the cursor**
  - As a single tuple if only one row matched
  - As a tuple of tuples if multiple rows matched

```
curs.execute('SELECT city FROM airport')
for name in curs:
    print('Airport name', name)
```

- **`fetchone()` returns the next tuple**
- **`fetchall()` method returns a tuple of tuples with all remaining rows**
- **`fetchmany(`*`size`*`)` method returns *`size`* tuples within a tuple**
- **All return `None` after all rows have been returned**

Example SQL Query

sql1.py

```
import sqlite3

connection = sqlite3.connect(r'C:\Course\1905\Data\airline.db')
curs = connection.cursor()
curs.execute('SELECT * FROM aircraft')
for line in curs:
    print(line)
connection.close()
```

```
(1, 'Canadian Regional Jet')
(2, 'Embraer Regional Jet')
(3, 'Airbus 319')
(4, 'Boeing 767')
```

Constructing a SELECT String

- **The SQL command must be a single string**
- **May be referenced by a variable**
  - Or variables concatenated

```
query = 'SELECT * FROM flights WHERE flightnum = 1587'

curs.execute(query)
```

Passing Arguments to SQL Statements

- **Statements contain fixed SQL syntax and parameters**
  - Values substituted from the arguments
  - `sqlite3` uses `?` for positional parameter
- **Other parameter symbols may be used for other database APIs**
  - `%s`, or *`column_name`*

sql2.py

```
craftnum = (2, )
apt = ('HNL', )

curs.execute('SELECT * FROM aircraft
                        WHERE aircraftcode = ?', craftnum)

curs.execute('SELECT * FROM airport
                       WHERE citycode = ?', apt)
```

Argument is a sequence

Placeholder for argument

Inserting a Row

- **Use the SQL `INSERT INTO` *`table`* `VALUES (...)` statement**
  - Values inserted are a tuple
  - Values must meet the database field constraints
- **Call the connection's `commit()` method to update the database storage**

sql3.py

```
newplane1 = (5, 'Blimp')
newplane2 = (6, 'Helicopter')

curs.execute('INSERT INTO aircraft VALUES (?, ?)',
             newplane1)
curs.execute('INSERT INTO aircraft VALUES (?, ?)',
             newplane2)

connection.commit()
```

Tuples

Assigned left to right

Updating Data

- **Use the SQL `UPDATE` *`table`* `SET ... WHERE ...` statement**
  - Modifies the fields specified by `SET`
    - For the rows specified by `WHERE`
- **Call the connection's `commit()` method to update the database storage**

sql4.py

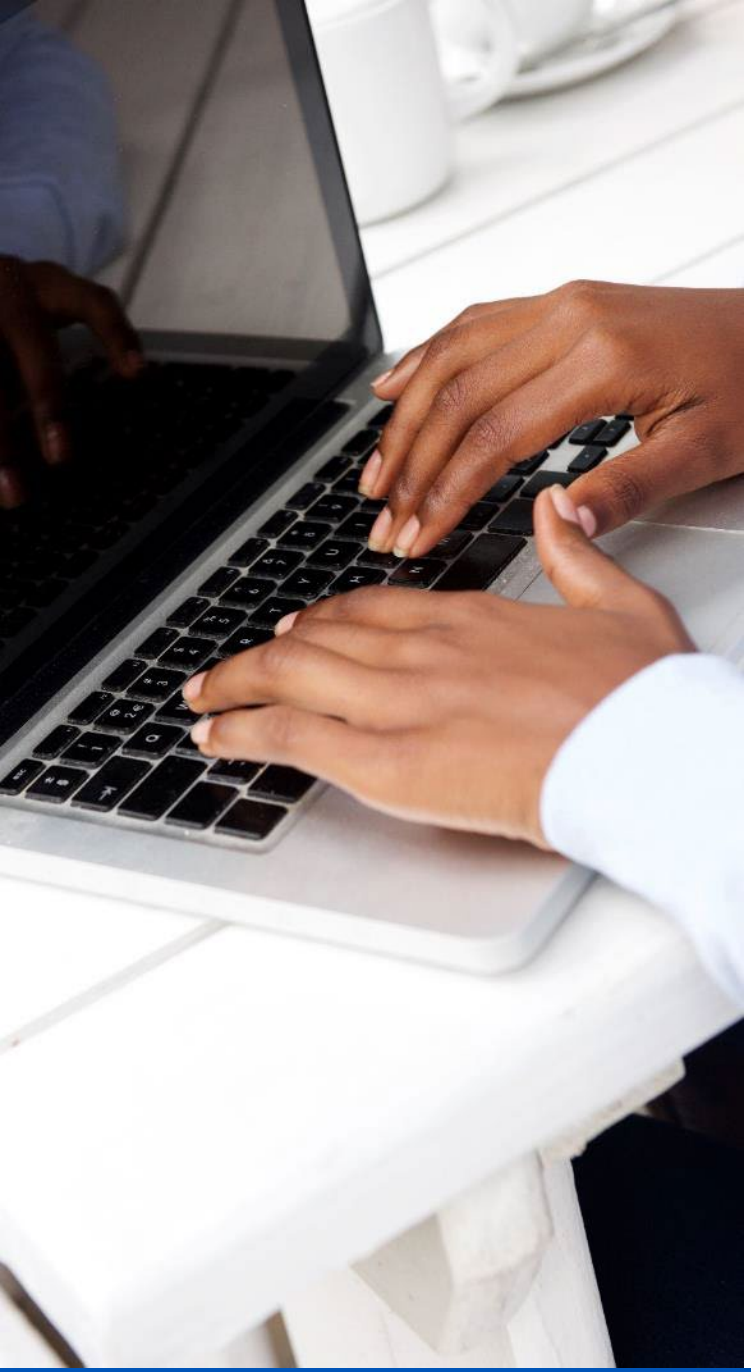
```
updateplane1 = (7, 'Blimp')
updateplane2 = ('Bell430', 6)

curs.execute('UPDATE aircraft SET aircraftcode = ?
              WHERE name = ?', updateplane1)

curs.execute('UPDATE aircraft SET name = ?
              WHERE aircraftcode = ?', updateplane2)

connection.commit()
```

Deleting Data

- Use the SQL `DELETE FROM table WHERE` statement
- Call the connection's `commit()` method to update the database storage

sql5.py

```
planecode = (6, )
planetype = ('Blimp', )

curs.execute('DELETE FROM aircraft
                     WHERE aircraftcode = ?', planecode)

curs.execute('DELETE FROM aircraft
                     WHERE name = ?', planetype)

connection.commit()
```



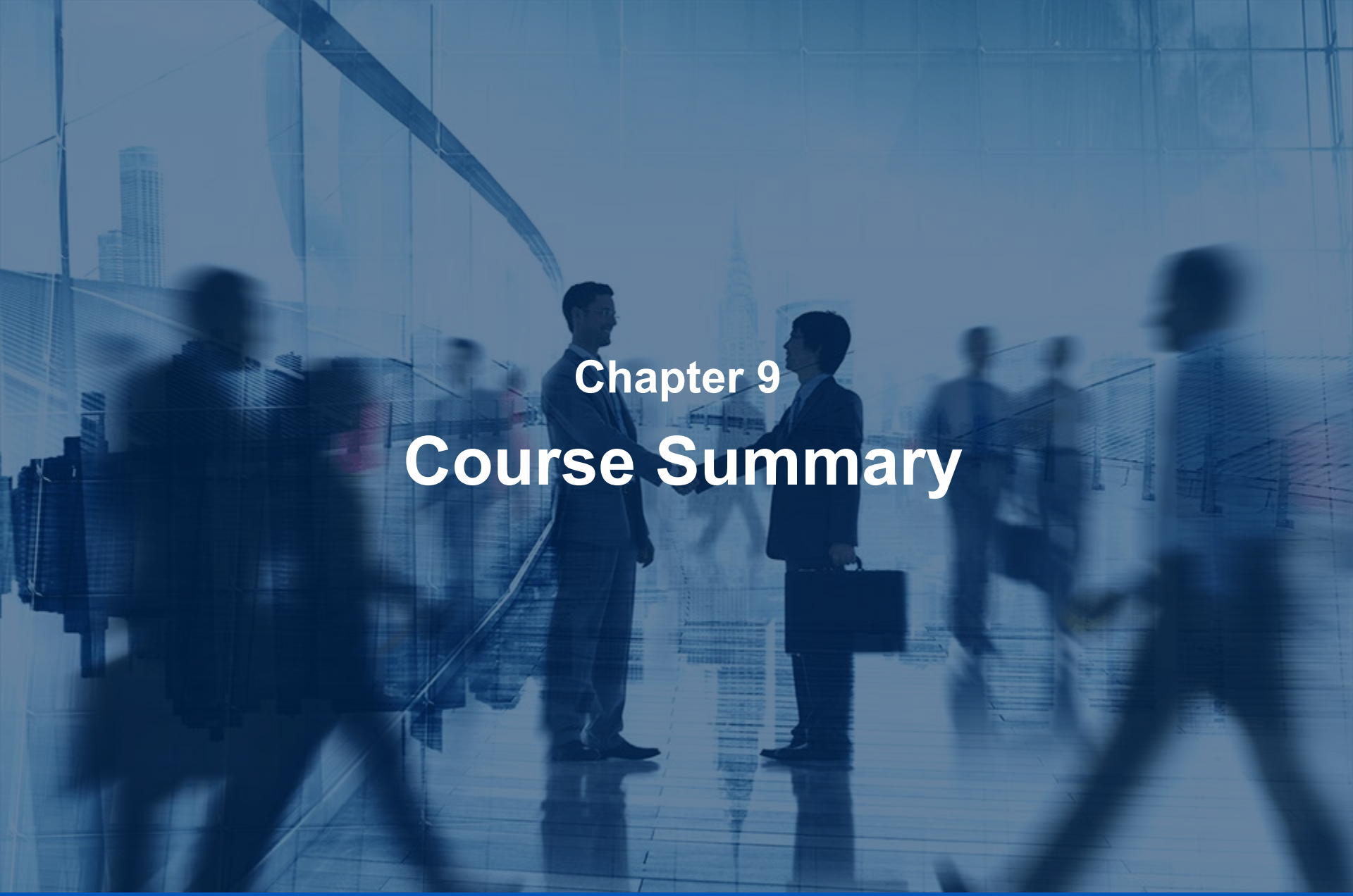
Hands-On Exercise 8.1

Exercise Manual

In your Exercise Manual, please refer to Hands-On Exercise 8.1: Accessing a Database

Objectives

- Access databases with Python using SQL



Chapter 9

Course Summary

LEARNING TREE™ INTERNATIONAL

Course Objectives

- Create, edit, and execute Python programs in PyCharm
- Use Python simple data types and collections of these types
- Control execution flow: conditional testing, loops, and exception handling
- Encapsulate code into reusable units with functions and modules
- Employ classes, inheritance, and polymorphism for an object-oriented approach
- Read and write data from files
- Query databases using SQL statements within a Python program