



Introduction to Python





Course Overview

The Core Python training course is designed to get you acquainted and up and running with the Python programming language.

The course begins with a quick look at why you would adopt Python, where Python excels, and how it compares to other object-oriented programming languages.





Course Overview

It then transitions into a nuts-and-bolts examination of key language features, concepts, and functionalities.

Along the way, you will learn how to apply OO programming concepts with Python, work with exceptions, and create modularity in your applications.





Course Objectives

After this course, you will be able to:

-  Install and configure your development environment to support Python
-  Create a basic stand-alone Python application
-  Perform basic text-processing functionality using Python
-  Create a modular application





Course Roadmap

- ◇ 1. Overview of Python
- ◇ 2. Installation and Setup
- ◇ 3. Python – The Basics
- ◇ 4. Data Structures
- ◇ 5. Exception Handling





Course Roadmap

- ◆ 6. Strings and Regular Expressions
- ◆ 7. Packages and Modules
- ◆ 8. OO Programming
- ◆ 9. Working with decorators
- ◆ 10. Developer Modules





1

Overview of Python

A really cool scripting language





Unit Objectives

After this lesson, you will be able to:

-  Explain what type of language Python is
-  Debate the pros and cons of using Python
-  Compare Python to other programming languages





1.1 What is Python





Python – Monty Python's Flying Circus







"Python is an **interpreted, object-oriented**, high-level programming language with **dynamic semantics**.

Its high-level built in data structures, combined with **dynamic typing** and **dynamic binding**, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together."

Further reading:
<https://www.python.org/doc/essays/blurb/>





Translation

INTERPRETED

Translates code (.py) to Bytecode (.pyc) for the Interpreter, which then compiles the code (.pyc) for the native machine.

OBJECT ORIENTED

A programming paradigm based on Objects, which are data structures that contain data and procedures.





Translation

DYNAMIC SEMANTICS

The dynamic semantics of a language defines what happens when you run a program.

DYNAMIC TYPING

You do not have to specify the variable type (e.g. int, num, long, etc). In Statically Typed languages, you do.

DYNAMIC BINDING

Like Dynamic Typing. Variable type determined at runtime.





Strong vs Dynamic Typing

Python is strongly, dynamically typed.

STRONG TYPING

Strong typing means that the type of a value doesn't suddenly change. A string containing only digits doesn't magically become a number, as may happen in Perl. Every change of type requires an explicit conversion.

DYNAMIC TYPING

Dynamic typing means that runtime objects (values) have a type, as opposed to static typing where variables have a type.





The Python Interpreter

There are four steps that Python Interpreter takes: lexing, parsing, compiling, and interpreting.

1. **Lexing** is breaking the line of code you just typed into tokens.
2. The **parser** takes those tokens and generates a structure that shows their relationship to each other (in this case, an Abstract Syntax Tree).





The Python Interpreter

3. The compiler then takes the Abstract Syntax Tree and turns it into one (or more) code objects.
4. Finally, the interpreter takes each code object and executes the code it represents.

Further reading:

<http://akaptur.com/blog/2013/11/15/introduction-to-the-python-interpreter/>





Is Python a scripting language?

“It's worth noting that languages are not interpreted or compiled, but rather language implementations either interpret or compile code.”

Further reading:

<http://programmers.stackexchange.com/questions/46137/what-is-the-main-difference-between-scripting-languages-and-programming-language>





Implementations

- ◇ CPython – default
- ◇ PyPy – JIT Compiler
- ◇ Jython – JVM

Further reading:

<https://wiki.python.org/moin/PythonImplementations>





Everything is an object

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute `__doc__`, which returns the doc string defined in the function's source code.

Further reading:

<https://docs.python.org/3.1/reference/datamodel.html>





The Zen of Python

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Further reading:

<https://www.python.org/dev/peps/pep-0020/>





Python has great quotes

“A Foolish Consistency is the Hobgoblin of Little Minds”

Further reading:

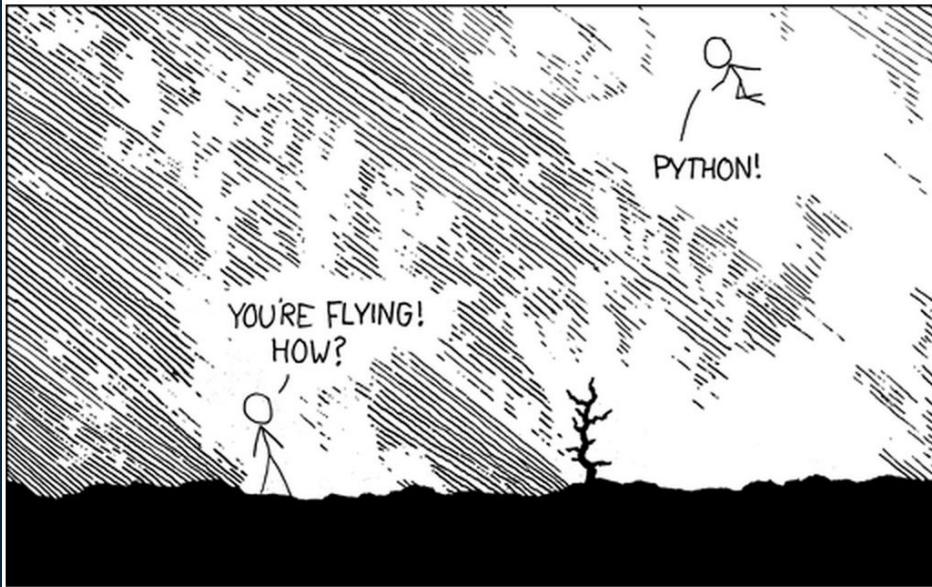
<https://www.python.org/dev/peps/pep-0008/>





1.2 Why use Python





I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
/ HELLO WORLD IS JUST
print "Hello, world!"

I DUNNO...
DYNAMIC TYPING?
WHITESPACE?
/ COME JOIN US!
PROGRAMMING IS FUN AGAIN!
IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?

I JUST TYPED
import antigravity
THAT'S IT? /
/ ... I ALSO SAMPLED
EVERYTHING IN THE
MEDICINE CABINET
FOR COMPARISON.
/ BUT I THINK THIS
IS THE PYTHON.





"Python has been an important part of Google since the beginning, and remains so as the system grows and evolves. Today dozens of Google engineers use Python, and we're looking for more people with skills in this language."

Peter Norvig, Google





"Python is fast enough for our site and allows us to produce maintainable features in record times, with a minimum of developers,"

Cuaong Do, Youtube





Pro's

- ◊ Easy to read
- ◊ Simplistic syntax
- ◊ Virtually no boilerplate

Further reading: <http://www.infoworld.com/article/2887974/application-development/a-developer-s-guide-to-the-pros-and-cons-of-python.html>





Python's main advantage

COLLABORATION

Because Python's syntax and style are easy to read, collaborating on a coding project inside a dev team is a easier and more enjoyable process.

You can more quickly intuit another dev's intent because their code is easier to read.





Hello, World!

Java

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, World");  
    }  
}
```

Python

```
print 'Hello, World'
```





Say hello!

Lab 1.1

PROBLEM

Your terminal has a desperate need to greet the world.

SUCCESS CRITERIA

Use Python to help your terminal express itself and say, “Hello, World!”





Go to Terminal

Step 1: Activate Python interactive mode

```
$ python
Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Step 2: Print “Hello, World!” to the screen

```
>>> print "Hello, World!"
Hello, World!
```





Lab Review

You learned how to run a basic program
in Python

All lab solutions can be found at:

<https://github.com/bengrunfeld/python-exercises>





"Code is more often read than it is written"

Guido Van Rossum, Python

Further reading:
<https://www.python.org/dev/peps/pep-0008/>





Con's

- ◊ Weak in mobile computing
- ◊ Python isn't in Web browsers
- ◊ Consider speed options available





1.3 Python vs other languages





Python vs Java

- ◇ Python is easier to read – **collaboration!**
- ◇ Java Hotspot is faster than PyPy
- ◇ Indentation vs braces
- ◇ Java is more portable – JVM

Further reading:

<https://blog.udemy.com/python-vs-java/>





Python vs Ruby

The difference is mostly cultural.

Ruby devs prize freedom and rapid change, even at the cost of breaking legacy code.

Python devs admire more stable yet slower releases over shiny new features.

Further reading:

<https://www.scriptrock.com/articles/python-vs-ruby>

<http://ruby-doc.org/docs/ruby-doc-bundle/FAQ/FAQ.html>





Python 2 vs Python 3

In 2008, Python 3 was released and it was announced the Python 2 would not be continued.

- ◇ Better unicode support
- ◇ Division with integers (e.g. $5/2$)

Further reading:

<http://blog.teamtreehouse.com/python-2-vs-python-3>

<http://www.infoworld.com/article/2619428/python/van-rossum--python-is-not-too-slow.html>





Review Questions

-  Is Python dynamically or statically typed?
-  What are some of the advantages of using Python?
-  Name two syntax differences between Python and Java





2

Installation and Configuration

“Primum non nocere” – First, do no harm





Unit Objectives

After this lesson, you will be able to:



Install Python on Mac & Windows 8



Isolate Python environments using VirtualEnv



Compile programs in the Shell





2.1 Installation





Installation on a Mac

Python **2.7.10** comes installed on all Mac's, so if you're a Mac user, you're all set!

If you have an older version of Python (i.e. <2.7.10) and want to upgrade, use the installer at:

<https://www.python.org/getit/>





Installation on a Windows 8

Installing Python 2.7.10 on Windows 8 is a bit of a lengthier process.

Follow the instructions at:

<http://stackoverflow.com/a/21373411/1676476>





2.2 Configuring the Python environment





Isolating your environment

PROBLEM #1:

If you have an app that depends on **Version 1.1.1** of a Library, but you have another app that relies on **Version 2.2.2** of the same Library, how do you install both without upgrading unintentionally?





Isolating your environment

PROBLEM #2:

If you go ahead and `pip install library`,
how do you stop `pip` from installing the
package in the global set of libraries,
overwriting other needed software?





Isolating your environment

SOLUTION:

Create a virtual environment for each app that installs dependencies in its own installation directories, and does not share libraries with other environments or overwrite general libraries.





Isolating your environment

IN A NUTSHELL:

Use **VirtualEnv** and **VirtualEnvWrapper**

Further reading:
<https://virtualenv.pypa.io/en/latest/>





Lab 2.1

Let's get virtual!

PROBLEM

Download the tools to isolate
Python libraries from each other
and from general libraries

SUCCESS CRITERIA

Install and use VirtualEnv and
VirtualEnvWrapper





Installation

Step 1: Install Pip

```
$ [sudo] easy_install pip
```

Step 2: Install VirtualEnv with Pip

```
$ [sudo] pip install virtualenv
```

Step 3: Install VirtualEnvWrapper with Pip

```
$ [sudo] pip install virtualenvwrapper
```





Installation

Step 4: Add the following to your .bashrc

```
export WORKON_HOME=$HOME/.virtualenvs  
export PROJECT_HOME=$HOME/Devel  
source /usr/local/bin/virtualenvwrapper.sh
```

Step 5: Source your .bashrc

```
source ~/.bashrc
```





Lab Review

Pip, VirtualEnv and VirtualEnvWrapper have all been installed on your machine.

Further reading:
<https://virtualenvwrapper.readthedocs.org/en/latest/install.html>



Lab 2.2

Use your Virtual Env

PROBLEM

Isolate downloaded Python libraries from the rest of the system

SUCCESS CRITERIA

Set up virtual environments for Python using VirtualEnvWrapper and download a package.





Usage

Step 1: Create a new virtual environment

```
$ mkvirtualenv tmp1  
New python executable in  
tmp3/bin/python  
Installing setuptools, pip, wheel...  
done.
```

Step 2: List your virtual environments

```
(tmp1)$ workon  
tmp1
```





Usage

Step 3: Create another Virtual Env

```
(tmp1) $ mkvirtualenv tmp2
```

Step 4: Install a package

```
(tmp2) $ pip install django
```

Step 5: List packages in tmp2

```
(tmp2) $ ls sitepackages
```





Usage

Step 6: Switch to tmp1

```
(tmp2) $ workon tmp1
```

Step 7: List the site packages there. Notice that Django is not present. It is only available in in tmp2.

```
(tmp1) $ lssitepackages
```





Usage

Step 8: Delete the tmp2 virtual environment

```
(tmp1) $ rmvirtualenv tmp2
```

Step 9: List virtual environments

```
(tmp1) $ workon  
tmp1
```

Step 10: Deactivate VirtualEnvWrapper

```
(tmp2) $ deactivate
```





Lab Review

You just learned how to created a virtual environment, populate it with a package in isolation, switch between environments, delete the environments you don't want, and exit VirtualEnvWrapper.

Note: It is best practice to use a separate virtual environment for every project.

Further reading:

<https://virtualenv.pypa.io/en/latest/>





2.3 Running a Python Program





Running a program

There are 3 ways to run a Python program.

1. Inside interactive mode
2. Using the `python` command in the Shell
3. Run the file directly using the Shebang `#!`





Lab 2.3

Up and running!

PROBLEM

You want Python to run your programs

SUCCESS CRITERIA

Programs execute successfully,
even if there are exceptions raised





Running a program

Step 1: Go into Python interactive mode

```
$ python  
<...output>  
>>> print "Hello, World!"  
Hello, World!
```

Step 2: Exit interactive mode

```
>>> exit()  
OR  
>>> Ctrl-D
```





Running a program

Step 3: Open `hello.py` in a text editor

```
print "Hello, World!"
```

Step 4: Run `hello.py` using the `python` command

```
$ python hello.py  
Hello, World!
```





Running a program

Step 5: Open `shebang.py` in a text editor

```
#! /usr/bin/env python  
print "Hash bang!"
```

Step 6: Give the file executable permission

```
$ chmod +x shebang.py
```





Running a program

Step 7: Give shebang.py executable permission

```
$ chmod +x shebang.py
```

Step 8: Add your current directory to \$PATH

```
$ pwd  
/Users/ben/  
PATH=$PATH:/Users/ben/
```





Running a program

Step 9: Run `shebang.py` without any command

```
$ shebang.py  
Hash Bang!
```





Lab Review

You just learned 3 different ways that a User can run a Python program from the command line.





Review Questions

-  What version of Python is installed by default on Mac?
-  Explain what a virtual environment is and why you should use one
-  Name the three ways you can run a Python program





3

Python – The Basics

"And now, for something different..."





Unit Objectives

After this lesson, you will be able to:

-  Understand basic syntax in Python
-  Set, use and manipulate variables
-  Create control structures and loops
-  Write function definitions





Mini-labs

A lot of the information in this unit does not lend itself to being used in a lab (e.g. declaring a variable).

Please follow along by inputting the shorter examples into Python's interactive mode. That way, when we get to the proper labs, you'll already have experienced using the basics.





3.1 Basic syntax





PEP 0008

PEP-8 is the style guide for Python Code, authored by Guido Van Rossum (BDFL), Barry Warsaw, and Nick Coghlan.

It is considered best practice to write Python code using the PEP-8 guidelines.

Further reading:

<https://www.python.org/dev/peps/pep-0008/>





Comments

Comments start with a **#** character, Unix style. Multiline comments simply all start with **#**.

```
# This is a comment
```

```
# This is a  
# multi-line comment
```

```
a = 5 * 9                      # 5 times 9
```





Indentation

Indentation is Python's way of grouping statements, as opposed to braces. Each line within a basic block must be indented by the same amount.

```
>>> b = 1
... while b < 10:
...     print b
...     a = b
...     b = a + b
```





Colons

If statements, loops, function definitions, and class definitions all use colons to declare the start of an indented block.

```
>>> b = 1
... while b < 10:
...     print b
...     a = b
...     b = a + b
```

Further reading:

<https://docs.python.org/2/faq/design.html#why-are-colons-required-for-the-if-while-def-class-statements>





Arithmetic operators

+	Addition	$5 + 5 = 10$
-	Subtraction	$10 - 5 = 5$
*	Multiplication	$5 * 2 = 10$
/	Division	$10 / 2 = 5$
%	Modulus	$10 \% 5 = 0$
**	Exponent	$3 ** 2 = 9$





Comparison operators

a = 5, b = 6

<code>==</code>	is equal to	<code>a == b</code> (false)
<code>!=</code>	not equal to	<code>a != b</code> (true)
<code>></code>	greater than	<code>a > b</code> (false)
<code><</code>	less than	<code>a < b</code> (true)
<code>>=</code>	greater than or equal to	<code>a >= b</code> (false)
<code><=</code>	less than or equal to	<code>a <= b</code> (true)





Python as a calculator

In interactive mode, if you type in a calculation, Python will evaluate it when you press enter

```
>>> 3 * 10 / 2 - 3  
12
```

But be careful...

```
>>> 5 / 2  
2
```

Wait, what?!





Python as a calculator

Python received an integer from you, and believes that you should receive one back. If you want to perform floating point arithmetic, at least one of the operands needs to be a float.

```
>>> 5.0 / 2  
2.5
```





3.2 Variables





Initializing variables

The equals sign = is used to assign a value to a variable

```
>>> ben = 35  
>>> mike = "A really cool guy"
```

TIP: To print out a variable's value, just type its name and press enter

```
>>> ben  
35
```





Initializing variables

A value can be assigned to several variables simultaneously

```
>>> x = y = z = 0
```

Multiple variables can be initialized on the same line, with different values

```
>>> a, b = 0, 1
```





Variable types

Python has four main numeric types:

- ◇ plain integers – 32 bit precision
- ◇ long integers – unlimited precision
- ◇ floating point numbers – double in C
- ◇ complex numbers – real & imaginary part





Casting variables

Use the `float()`, `int()`, or `long()` functions to cast a variable to the value of that type

```
>>> a,b,c = 5,10.5,15
>>> float(a)
5.0

>>> int(b)
10.5

>>> long(c)
15L
```





Strings

Assign strings with `=`, or just type them out

```
>>> "Good afternoon"  
'Good afternoon'  
  
>>> 'Good evening'  
'Good evening'  
  
>>> "'hello', he said"  
"'hello', he said"  
  
>>> 'won\'t you stay for dinner?'  
"won't you stay for dinner?"
```





Concatenation & repetition

Strings can be concatenated with `+` repeated with `*`

```
>>> a = "Blessed are the "
>>> b = "Cheesemakers"

>>> a + b
'Blessed are the Cheesemakers'

>>> a + b*2
'Blessed are the CheesemakersCheesemakers'
```





Multi-line strings

String literals can span multiple lines in several ways. Continuation lines can be used, with a backslash \ as the last character on the line indicating that the next line is a logical continuation of the line:

```
>>> great_quote = "Blessed are \  
... the Cheesemakers  
  
>>> print great_quote  
'Blessed are the Cheesemakers'
```





Multi-line strings

Strings can be surrounded in a pair of matching triple-quotes: """ or '''. End of lines do not need to be escaped when using triple-quotes, but they will be included in the string.

```
>>> great_quote = """Blessed are  
... the Cheesemakers"""

>>> print great_quote
Blessed are
the Cheesemakers
```





3.3 If statements





If statements

There can be zero or more `elif` parts, and using `else` is optional.

```
>>> x = 5
>>> if x > 4:
...     print "X is greater than four"
... elif x < 4:
...     print "X is less than four"
... else:
...     print "X is something else"
```

X is greater than four





and & or

In Python, we use `and` and `or` to perform Boolean operations, not `&&` or `||`

```
>>> a = 10
... if a == 5 or a == 10:
...     print 'yes'
yes
```

```
>>> a = 10
... if a % 10 == 0 and a % 5 == 0:
...     print 'yes'
yes
```





not

To negate a statement, use **not**

```
>>> a = 10
... if not a == 10:
...     print 'a does not equal 10'
... else:
...     print 'a equals 10'
```

```
a equals 10
```



3.4 Loops





The while loop

The while loop will continue looping as long as the condition it's testing for evaluates to true.

```
>>> b = 1  
>>> while b < 10:  
...     print b  
...     a = b  
...     b = a + b
```

```
1  
2  
4  
8
```





The for loop

The for statement in Python differs a bit from what you may be used to in Java.

Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as Java), Python's for statement iterates over the **items of any sequence** (a list or a string), in the order that they appear in the sequence.





The for loop

We'll create a sequence for the for loop to iterate over using the `range` statement.

```
>>> range(5)
[0, 1, 2, 3, 4]

>>> for a in range(5):
...     print a

0
1
2
3
4
```





The for loop

But we could just have easily have used strings

```
>>> greeting = "Hello"  
  
>>> for greet in greeting:  
...     print greet
```

```
H  
e  
l  
l  
o
```





The break statement

The break statement, borrowed from C, breaks out of the smallest enclosing for or while loop.

```
>>> for a in range(5):
...     if a > 2:
...         break
...     print a
```

```
0
1
2
```





The continue statement

The `continue` statement, also borrowed from C, continues with the next iteration of the loop.

```
>>> for a in range(5):  
...     if a == 2:  
...         continue  
...     print a
```

```
0  
1  
3  
4
```



Lab 3.4

Yep, it's fizzbuzz

PROBLEM

Print the numbers 0 - 100. For every multiple of 3, print “fizz”. For every multiple of 5, print “buzz”. For every multiple of both 3 and 5, print fizzbuzz”

SUCCESS CRITERIA

The program will replace the multiples as stated above, and will be runnable without use of the `python` command



Fizzbuzz

Step 1: Create a new Python file

```
$ vi fizzbuzz.py
```

Step 2: Add a Shebang at the top of the file

```
#! /usr/bin/env python
```

Step 3: Use range() in your for statement

```
for fb in range(101):
```





Fizzbuzz

Step 4: Test for the conditions you're interested in. Remember to indent your code

```
for fb in range(101):  
    if fb % 3 == 0 and fb % 5 == 0:  
        print "fizzbuzz"  
        continue
```

Step 5: Test for the other conditions

```
elif fb % 3 == 0:  
    print "fizz"  
    continue
```





Fizzbuzz

Step 6: Print out the current index if all other conditions are false

```
print fb
```





Fizzbuzz

Solution

```
#! /usr/bin/env python

for fb in range(100):
    if fb % 3 == 0 and fb % 5 == 0:
        print "fizzbuzz"
        continue
    elif fb % 3 == 0:
        print "fizz"
        continue
    elif fb % 5 == 0:
        print "buzz"
        continue
    print fb
```





Lab Review

From this exercise, you learned how to use loops in Python, use conditionals with boolean logic, and use the range statement to generate a sequence. Also, fizzbuzz is asked at almost every coding interview ;)



3.5 Functions





Defining functions

To define a function:

```
def my_function_name(params) :
```

The keyword **def** introduces a function definition. It must be followed by the **function name** and the parenthesized list of formal **parameters**. The statements that form the body of the function start at the next line, and must be indented.





Docstrings

The first statement of the function body can optionally be a string literal. This string literal is the function's documentation string, or **docstring**.

```
def return_root_name(params):  
    """Returns the root name of dir"""
```

Further reading:
<https://www.python.org/dev/peps/pep-0257/>





Docstrings

There are tools which use docstrings to automatically produce online or printed documentation (e.g. Sphinx), or to let the user interactively browse through code.

It's good practice to write docstrings for your functions, so that other programmers can easily understand what your intention was.

Further reading:

<http://sphinx-doc.org/>





Using parameters

A function may be declared with multiple parameters, which must be used in the function call.

```
def sum_of_params(a, b):  
    print a + b  
  
sum_of_params(3, 5)  
8
```

Further reading:

<https://docs.python.org/2/tutorial/controlflow.html#define-functions>





Default argument values

A function can be declared with default values set for its parameters. This way, the function can be called with fewer arguments than it is defined to allow.

```
def sum_of_params(a, b=5):  
    print a + b  
  
sum_of_params(3)  
5
```

Further reading:

<https://docs.python.org/2/tutorial/controlflow.html#default-argument-values>





Calling the function

To call the function, simply use its name, followed by parentheses which are filled with the correct number of parameters.

```
def breakfast(a, b, c, d):  
    print a, b, c, "and", d  
  
breakfast("spam", "spam", "sausage", "spam")  
spam spam sausage and spam
```



Lab 3.5

Apple Pi

PROBLEM

Your app needs to calculate the sum of two separate inputs, and then multiply the result by Pi.

SUCCESS CRITERIA

The program will perform the calc inside a function, which will use a docstring to document your intent. It will be runnable from the CLI.



Using functions

Step 1: Create a new Python file

```
$ vi apple.py
```

Step 2: Add a Shebang at the top of the file

```
#! /usr/bin/env python
```

Step 3: Declare a function with 2 params

```
def apple_pi(a, b):
```





Using functions

Step 4: Write a Docstring to document your intent

```
"""Return sum of the inputs multiplied by  
Pi"""
```

Step 5: Perform the calculations and return the result

```
def apply_pi(a, b):  
    """Docstring"""  
  
    c = a + b  
    return c * 3.141592
```





Using functions

Step 6: Call your function with params and store the result

```
result = apple_pi(3, 5)
```

Step 7: Print out result

```
print result
```

Step 8: Chmod your file and make sure PWD is in \$PATH

```
chmod +x apple.py
```





Using functions

Step 9: Run the file from the command line
and the result should print to the terminal

```
$ apple.py  
47.12388
```





Lab Review

Functions are the modular building blocks of nearly all Python programs. By applying best practice in the form of Docstrings, you ensure that other programmers will understand your intent.





Multiple arguments

Functions can be declared with multiple arguments. We use `*` to denote a list the the function definition, and `**` to denote a dict.

```
>>> def breakfast(a, b, *c):
...     print a, b, c

>>> breakfast('spam', 'sausage', ['eggs', 'bacon'])
spam sausage ['eggs', 'bacon']
```





Review Questions

-  What syntax does Python use to declare the the following code should be indented?
-  How do you write `else if` in Python?
-  What does Python's `for` loop iterate over?
-  What are 2 ways to create a multi-line string?
-  What word is used to declare a function?



4

Data Structures

"My hovercraft is full of eels!"

Hungarian with translating book at the Tobacconist [spam]





Unit Objectives

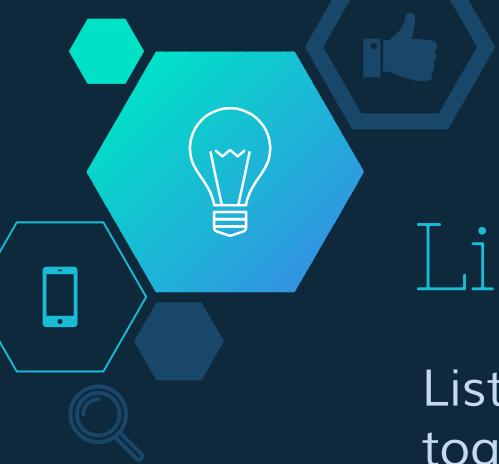
After this lesson, you will be able to:

-  Explain the difference between the Python types
-  Employ lists, tuples, dicts and list comprehensions in your code
-  Understand the particular implementations of each Type



4.1 Lists





Lists

Lists are a data type that are used to group together other values. Lists can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```
>>> list = ['spam', 'eggs', 100, 1234]  
>>> list  
['spam', 'eggs', 100, 1234]  
  
>>> list[0]  
'spam'
```





List methods

Append x to the end of list

```
list.append(x)
```

Extend the list by appending another list

```
list.extend(L)
```

Insert item x at index i

```
list.insert(i, x)
```





List methods

Remove the first item of value x from list

```
list.remove(x)
```

Sort the items of the list

```
list.sort()
```

Reverse the elements of the list in place

```
list.reverse()
```





List methods

Return the number of times `x` appears in `list`

```
list.count(x)
```

Remove the item at the given position in `list`,
and return it. If no index is specified, `pop()`
removes and returns the last item in the list.

```
list.pop(i)
```

Further reading:
<https://docs.python.org/2/tutorial/datastructures.html>





Using lists as stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”).

To add an item to the top of the stack, use `list.append()`. To retrieve an item from the top of the stack, use `list.pop()` without an explicit index.





Using lists as stacks

```
>>> a = [5, 6]
>>> a.append(7)
```

```
>>> a
[5, 6, 7]
```

```
>>> a.pop()
7
```

```
>>> a
[5, 6]
```





Using lists as queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out"), however lists are not efficient for this purpose.

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends.





Lab 4.1

Shopping list

PROBLEM

You need your desk-neighbor to pick you up some things from the store. Write them a list, then make some changes.

SUCCESS CRITERIA

Your desk-neighbors items will be added to your list in order. Changes to the list will be made in the correct place. You must bulk add several items, and the final list must be sorted.





Advanced students

All additions and changes to your list must be made via a function call. The function must adhere to PEP-8 naming standards, and must contain a Docstring.





Lab Review

Lists are similar to arrays, which are heavily used in all programming. Efficient and correct usage of Lists will give an incredible speed boost to the app you're working on, and eventually in more readable code.



4.2 Tuples





Tuples

Lists and Strings are both of data type "sequence". Tuples are also of type "sequence", although Tuples have different rules to the others.

```
>>> t = (111, 222, 'hello!')  
  
>>> t  
(111, 222, 'hello!')  
  
>>> t[0]  
111
```





Tuples – rules

Tuples may be nested.

```
>>> t = (1, 2, (4, 5))
```

```
>>> t  
(1, 2, (4, 5))
```

```
>>> t[2]  
(4, 5)
```





Tuples – rules

Tuples are immutable.

```
>>> t = ("welcome", 2, 3)
```

```
>>> t[0] = "hello"
```

```
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: 'tuple' object does not support  
item assignment
```





Tuples – rules

...but tuples can contain mutable objects.

```
>>> t = (1, 2, [3, 4])
```

```
>>> t  
[1, 2, [3, 4]]
```

```
>>> t[2][0] = 5
```

```
>>> t  
[1, 2, [5, 4]]
```





Tuples – rules

To create an empty tuple, just use empty parens

```
>>> empty = ()
```

To create a tuple with only 1 item, use a comma

```
>>> singleton = ("hello",)
```





When to use tuples

- ◇ Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list.
- ◇ Tuples make your code safer by "write-protecting" data that does not need to be changed.



Lab 4.2

Tuple iteration

PROBLEM

A data set has been provided to you and you must iterate over the data and concatenate a string to the end of each item

SUCCESS CRITERIA

Each item in the data will be injected into a string, which gives linguistic context to data.
Run the Python file from the shell.



Tuple lab

The data represents this week's lottery numbers

```
32, 81, 74, 16, 25, 90, 53
```

Place the data in a tuple, then iterate over it using a for loop.

Print out each item with a message indicating its numeric order.

E.g. "The first number is 32"





Lab tips

Search for a way to use the `format` function to inject a variable into a string.

Search for a way to print out an index in a for loop using `enumerate`.





Lab Review

Tuples provide a way to write-protect your data, so that it is not accidentally or intentionally changed by others. This exercise showed you a use case where it would be appropriate to use tuples.



4.3 Sets





Sets

A set is an unordered collection with no duplicate elements.

```
>>> names = ['adam', 'ben', 'charles']

>>> unordered = set(names)

>>> unordered
set(['charles', 'ben', 'adam'])
```





Sets

Sets can be created by using the `set()` function, or by using curly braces {}.

```
>>> names = ['adam', 'ben', 'charles']
>>> unordered = set(names)

>>> unordered
set(['charles', 'ben', 'adam'])

>>> no_order = {'adam', 'ben', 'charles'}

>>> no_order
set(['charles', 'ben', 'adam'])
```





Set operations

Set objects support mathematical operations like union, intersection, difference, and symmetric difference.

```
>>> s = set('abc')
>>> t = set('bcd')

>>> s
set(['a', 'c', 'b'])

>>> t
set(['c', 'b', 'd'])
```





Set operations

Show items that are in s, but not in t

```
>>> s - t  
set(['a'])
```

Show items that are either in s or t

```
>>> s | t  
set(['a', 'c', 'b', 'd'])
```





Set operations

Show items that are in both s and t

```
>>> s & t  
set(['c', 'b'])
```

Show only items that are NOT in both s and t

```
>>> s ^ t  
set(['a', 'd'])
```



Lab 4.3

Set operations

PROBLEM

The Police department has supplied you with the member list of two dangerous criminal gangs. They need you to filter the data.

SUCCESS CRITERIA

Output which members belong to both gangs by injecting their names into a string that explains their dual membership.



Set lab

Here is the data the Police department provided you

```
bad_a_listers = ['Sean Connery', 'Matt Damon',  
'Christina Aguilera', 'Brad Pitt', 'Liam Neeson']
```

```
bad_b_listers = ['Rick Moranis', 'Matt Damon',  
'Bruce Campbell', 'Brad Pitt', 'Pierce Brosnan']
```





Lab tips

1. Move the data into sets.
2. Perform the operation.
3. Iterate over the results
4. Inject the appropriate value into a string





Lab Review

Sets give you the ability to efficiently filter unique data via comparative analysis, e.g. member records. Quickly eliminate the data you don't want, or create new lists that only contain the items you want.





4.4 Dictionaries





Dictionaries

Dictionaries are indexed by keys, which can be any immutable type. Strings and numbers can be used as keys.

```
>>> a = dict([('name', 'ben'), ('age', 35)])  
  
>>> a  
{'age': 35, 'name': 'ben'}
```





Dictionaries

Dictionaries can be created with curly braces {} or with the dict() function.

```
>>> b = {'age': 20, 'name': 'charles'}
>>> b
{'age': 20, 'name': 'charles'}
```



```
>>> a = dict([('name', 'ben'), ('age', 35)])
>>> a
{'age': 35, 'name': 'ben'}
```





ictionaries – rules

- ◆ Dictionaries are called "associative arrays" in other languages
- ◆ Keys must be unique
- ◆ A pair of braces creates an empty dictionary: {}
- ◆ It is possible to delete a key:value pair with `del`
- ◆ If you store a value using a key that is already in use, the value is overwritten





Dictionary functions

Returns a list of all the keys used in the dictionary

```
>>> a.keys()  
['age', 'name']
```

Returns a list of all the values in the dictionary

```
>>> a.values()  
[35, 'ben']
```





Lab 4.4

Genetics

PROBLEM

The bureau of Science believe they've found the protein (GAA) responsible for cancer, but they need you to test for its existence in a gene.

SUCCESS CRITERIA

Programmatically check a dictionary for the existence of the protein, and if it is there, remove it from the dict. Output the clean dict.





Dict lab

Here is the data the Science bureau provided you:

```
proteins = {'GCA': 'stable', 'TGC': 'stable',
'TGC': 'stable', 'GAA': 'volatile', 'ACT':
'stable', 'TTG': 'stable', 'GCT': 'stable'}
```





Lab Review

Dicts, by their nature, map to JSON objects and other similar data structures. By learning to manipulate dicts, you'll be able to use Python to work with a wide range of data structures in the future.





Review Questions

-  In what situation would you use a Tuple?
-  Should you use Lists as both stacks and queues?
-  If I wanted to find what items were in one list but not in another, what code would I use?
-  What other data structure is a Dict similar to?





5

Built-in functions

"My hovercraft is full of eels!"

Hungarian with translating book at the Tobacconist [spam]





Unit Objectives

After this lesson, you will be able to:



Understand the concept of Built-ins



Use Python Built-ins to make your code more complex



Choose the appropriate Built-in to use in certain situations



5.1 Type





Type

Use the `type()` built-in to ascertain the type of an object. Whenever in doubt, use `type`!

```
>>> a = {'name': 'ben', 'age': 35}

>>> a
{'age': 35, 'name': 'ben'}

>>> type(a)
<type 'dict'>
```



5.2 Slicing





Slicing

Slicing returns the items between the specified indices in an object. Slicing can be used for anything that supports numeric indexing. E.g. Strings, Lists, Tuples, etc

```
>>> a = ['tim', 32, 'bob', 27.6, 'mark']
```

```
>>> a[2:4]  
['bob', 27.6]
```





Slicing

Slicing returns a shallow copy of a list, which means that every slice returns an object which has a new address in memory, but its elements would have the same addresses that elements of source list have.



Slice notation

```
a[i:p]      #items i through p  
a[i:]       #items i until end of the object  
a[:p]        #items from the start of object through  
p  
a[:]         #a shallow copy of the whole object  
  
a[i:p:step] # i through p, using a step
```





5.3 Unicode object





Unicode object

In Python 3, all strings are Unicode by default, but in Python 2, we must explicitly define them.

```
>>> a = unicode('hello there')
>>> a
u'hello there'

>>> a.encode('utf-8')
'hello there'
```



5.4 Range





Range

The range statement generates lists containing arithmetic progressions. It has a finishing point, and optionally a starting point and a step.

```
>>> range(5)
[0, 1, 2, 3, 4]

>>> range(5, 10)
[5, 6, 7, 8, 9]

>>> range(3, 10, 2)
[3, 5, 7, 9]
```





Range

Range can be combined with other functions, like len() to provide useful results.

```
>>> a = ['life', 'is', 'good']

>>> for i in range(len(a)):
...     print i, a[i]

0 life
1 is
2 good
```





5.5 Pass





Pass

The Pass statement creates an empty placeholder that can be used to indicate the the appropriate code hasn't been written yet.

```
for r in range(10):  
    pass  
  
def sell_item(price):  
    pass
```





5.6 In, not in





In, not in

The `in` keyword tests whether a sequence contains a certain value. The `not` keyword negates `in`.

```
>>> a  
['life', 'is', 'good']
```

```
>>> 'life' in a  
True
```

```
>>> 'taxes' not in a  
True
```



5.7 Del





Del

Use `del` to delete variables, items from a list or dict, or even slices.

```
>>> a  
['life', 'is', 'very', 'good']  
  
>>> del a  
>>> del a[:2]  
  
>>> b = {'name': 'ben', 'age': 35}  
>>> del b['age']
```





Dict lab

Here is the data the Science bureau provided you:

```
proteins = {'GCA': 'stable', 'TGC': 'stable',
'TGC': 'stable', 'GAA': 'volatile', 'ACT':
'stable', 'TTG': 'stable', 'GCT': 'stable'}
```





Lab Review

Dicts, by their nature, map to JSON objects and other similar data structures. By learning to manipulate dicts, you'll be able to use Python to work with a wide range of data structures in the future.





Review Questions

-  How do you leave a placeholder in your code?
-  Create an arithmetic progression with a negative step
-  Delete the 3rd item in this list `a = [4, 6, 9]`
-  What function do we use to translate unicode to UTF-8?



6

Advanced looping





Unit Objectives

After this lesson, you will be able to:



Loop over a variety of data structures



Loop over multiple sequences concurrently



Loop over sorted or reversed items





6.1 List comprehensions





List comprehensions

List comprehensions provide a concise way to create lists. A common application is to make a new list where each element is the result of some operations applied to each member of another sequence or iterable.

```
>>> squares = [x**2 for x in range(10) if x > 3]  
  
>>> squares  
[16, 25, 36, 49, 64, 81]
```





6.2 Enumerate





Enumerate

Returns an enumerate object, which you can use as an index in a loop

```
>>> guests = ['adam', 'ben', 'charles']

>>> for i, guest in enumerate(guests):
...     print i, guest

0 adam
1 ben
2 charles
```





6.3 Loop over multiple lists





Multiple lists

You can loop over multiple sequences with `zip()`

```
>>> keys = ['name', 'quest', 'favorite color']
>>> values = ['lancelot', 'the holy grail', 'blue']

>>> for key, val in zip(keys, values):
...     print 'My {0} is {1}'.format(key, val)
```



6.1 Sorted and reverse





Sorted and Reverse

You can loop over a sorted or reversed sequence
with `sorted()` and `reversed()`

```
>>> attributes = ['name', 'quest', 'favorite color']

>>> for attr in sorted(attributes):
...     print attr

>>> for attr in reversed(attributes):
...     print attr
```



6.1 Iteritems





Iteritems

Return an iterator over the dictionary's (key, value) pairs.

```
>>> menu = {"starters": "scallops", "mains": "steak",
...           "dessert": "cake"}  
  
>>> for key, val in menu.iteritems():  
...       print key, val  
  
starters scallops  
dessert cake  
mains steak
```





Review Questions

-  How do you loop using an index in a list?
-  What does `iteritems()` do?
-  Is it possible to loop over multiple lists simultaneously? If so, how?
-  Explain how a list comprehension works



A decorative border of hexagonal icons surrounds the central content. The icons include: a lightbulb (top left), a thumbs-up (top center), a network graph (middle left), a smartphone (bottom left), a magnifying glass (bottom center), a gear (bottom right), a speech bubble (bottom left), and a small teal hexagon (bottom right).

7

Strings and Regex





Unit Objectives

After this lesson, you will be able to:



Perform string interpolation



Use StringIO to write to and read from the buffer



Execute regex actions on strings





7.1 String interpolation





Interpolation

To inject the values of variables into a string, use `format()`. It is less memory heavy than `+`.

```
>>> a = "apple"  
>>> b = "banana"  
  
>>> "Get me an {}-{} fritter".format(a, b)  
"Get me an apple-banana fritter"
```





Interpolation

To inject the values of variables into a string, use `format()`. It is less memory heavy than `+`.

```
>>> a = "apple"  
>>> b = "banana"  
  
>>> "Get me an {}-{} fritter".format(a, b)  
"Get me an apple-banana fritter"
```





Interpolation

The `format` command can take positional arguments.

```
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs
```

```
>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```





Interpolation

It can also identify values in the list with keyword arguments.

```
>>> print 'This {food} is {adjective}.'.format(food='chocolate,  
adjective='delicious')
```

This chocolate is delicious.





Interpolation

It can also identify values in the list with keyword arguments.

```
>>> print 'The story of {0} and {other}.'.format('John', other='Ringo')
The story of John and Ringo.
```



7.2 Str and repr





Str & repr

To convert any value to a string, use the `repr()` or `str()` functions.





Str

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax).



7.2 StringIO





StringIO

In some cases, you may want to write a string to a memory file (aka. buffer), add to it, and then later read from it.

```
import StringIO

file = StringIO.StringIO()
file.write("Blessed are the cheesemakers ")
file.write("Did he say cheese makers?")

print file.getvalue()          #read the file
file.close()                  #close the file
```





7.3 String operations





String operations

Test if a string contains a substring

```
>>> a = "bunch of coconuts"  
>>> a in "I've got a lovely bunch of coconuts"  
True
```

Convert to upper case

```
>>> "good morning, sir".upper()  
'GOOD MORNING, SIR'
```

Convert to lower case

```
>>> "GOOD MORNING, SIR".lower()  
'good morning, sir'
```





String operations

Count the occurrences of a character or sequence

```
>>> 'good morning, sir'.count('o')  
3
```

Search a string for the first occurrence of substr

```
>>> 'good morning, sir'.find('morn')  
5
```

Convert to lower case

```
>>> "GOOD MORNING, SIR".lower()  
'good morning, sir'
```





String operations

Replace all occurrences of substr with another string

```
>>> Good Good Birthday.replace('Good', 'Happy')  
Happy Happy Birthday
```

Cast to string

```
>>> str(9999)  
'9999'
```





7.4 Regular Expressions





Regular expressions

We use regular expressions to match patterns inside of Strings.

The `re` module provides Perl-style regular expression patterns which you can use to match expressions.

```
import re
```





Regex – use raw strings

PROBLEM

Python interprets special characters inside of regular strings. E.g. `\n` means new line. But the `re` module also uses the `\` character and gives it special meaning, so there can often be a conflict.





Regex – use raw strings

SOLUTION

Raw strings strip away all meaning from a string, so that the compiler doesn't misunderstand your intention. Prefix your string with **r** to turn it into a raw string.

```
>>> r'this is a raw stri\ng'
```





Perl Regex Syntax

^	Beginning of string
\$	End of string
.	Any character except new line
*	Match 0 or more times
+	Match 1 or more times
?	Match 0 or 1 times





Perl Regex Syntax

	Alternative
()	Grouping
[]	Set of characters
{}	Repetition modifier
/	Quote or special character





Perl Regex Syntax

a^*	Zero or more a's
a^+	One or more a's
$a^?$	Zero or one a's
$a^{\{m\}}$	Exactly m a's
$a^{\{m,\}}$	At least m a's
$a^{\{m,n\}}$	At least m, but at most n a's





Perl Regex Syntax

\w	any word character
\W	any non-word character
\s	any whitespace character
\S	non whitespace character
\d	any digit character
\D	any non-digit character





Regex – match

Match works by finding matches at the beginning of a string.

```
>>> re.match(r'cat', 'cats dogs')  
<_sre.SRE_Match object at 0x10ab9cb28>
```

```
>>> re.match(r'cat', 'dogs cats')
```





Regex – valid syntax

Any valid Perl-based regex syntax can be used.

```
person = "John Doe. 35."  
  
result = re.match(r'\w+ \w+. \d+. ',  
person)  
  
result.group(0)
```





Regex – group

When called with 0 as its argument, group will return the pattern matched by the query

```
result = re.match(r'cat', 'cats dogs')  
  
result.group(0)  
'cat'
```





Regex – search

Search is not restricted to the beginning of a string, so it will find a match anywhere, but it will stop searching after it has found the first occurrence of the pattern.

```
result = re.search(r'cat', 'dogs cats')  
  
result.group(0)  
'cat'
```





Regex –.findall

Finds all occurrences of the pattern in a string, and returns a list matching all patterns.

```
>>> re.findall(r'spam', 'spam eggs spam')  
['spam', 'spam']
```





Regex – match objects

Match and Search return a match object. This object contains information about the result, such as the starting and finished indices of the found pattern.

```
>>> match = re.search(r'dog', 'dog cat dog')  
  
>>> match.start()  
0  
  
>>> match.end()  
3
```





Regex – grouping

Grouping enables us to target certain parts of the regex match, and then work with them individually.

```
person = "John Doe. 35."  
  
result = re.match(r'(\w+) (\w+). (\d+)',  
person)  
  
result.group(0)
```





Regex – named groups

Python also lets us name the groups that we define, for ease of use with larger strings.

```
person = "John Doe. 35."  
  
result = re.match(r'(?P<first>\w+) (?P<last>\w+). (?P<age>\d+)', person)  
  
result.group('age')  
'35'
```





Regex – compile

A cleaner way to work is to compile your pattern into a regular expression object. This way, it can be reused easily, and other functions become available to it.

```
regex_obj = re.compile(r'\w+')

regex_obj
<_sre.SRE_Pattern object at 0x108c41ae0>
```





Regex – search & replace

Use `sub()` to replace every occurrence of a pattern inside a string.

```
>>> a = 'john doe'  
>>> obj = re.compile(r'jo\w+')  
  
>>> obj.sub('dave', a)  
'dave doe'
```





Review Questions

-  What Python re function do you use to match a pattern at the beginning of a string?
-  Which options is faster to inject a value into a string? format() or +? Please explain.
-  How does re.compile work? What does it offer that the alternatives don't?
-  How would you find the first occurrence of a substring?



The slide features a decorative border on the left side composed of various hexagonal icons. These icons include a lightbulb, a thumbs-up, a network graph, a smartphone, a magnifying glass, a gear, and a speech bubble. A large, semi-transparent teal hexagon is centered over the number 8.

8

Exception Handling

"My hovercraft is full of eels!"

Hungarian with translating book at the Tobacconist [spam]





Unit Objectives

After this lesson, you will be able to:



Handle exceptions



Raise specific exceptions



Define your own custom exceptions





8.1 Error types





Error types

There are (at least) two distinguishable kinds of errors: syntax errors and exceptions. Syntax errors occur during the parsing stage, while Exceptions occur later during the compilation phase.





Syntax errors

When your code is syntactically incorrect and you run it, the parser repeats the offending line, and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. File name and line number are printed for convenience.

```
>>> a = 'hello"  
File "<stdin>", line 1  
     a = 'hello"  
          ^  
  
SyntaxError: EOL while scanning string  
literal
```



8.2 Exceptions





Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal, because you, and your program, can handle them.





The try statement

1. First, the try clause (the statement(s) between the try and except keywords) is executed.
2. If no exception occurs, the except clause is skipped and execution of the try statement is finished.





The try statement

3. If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.





The try statement

4. If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.





The try statement

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed.





Error handling demo

```
>>> try:  
...     10 / 0  
... except ZeroDivisionError as err:  
...     print "ZeroDivisionError"  
  
ZeroDivisionError
```



Else

The try...except statement has an optional **else** clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception.

```
try:  
    10 / 0  
except ZeroDivisionError as err:  
    print "ZeroDivisionError"  
else:  
    print "No errors were encountered"
```



Finally

The try statement has another optional clause, **finally**, which is intended to define clean-up actions that must be executed under all circumstances.

```
try:  
    raise KeyboardInterrupt  
finally:  
    print 'thats all folks'  
  
thats all folks
```



Raising exceptions

The `raise` statement allows the programmer to force a specified exception to occur.

```
>>> raise NameError("Well that's a silly name")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: Well that's a silly name
```





Exceptions arguments

Arguments may be passed to an Exception, and then read from the Exception during handling.

```
>>> try:  
...      raise Exception('spam', 'eggs')  
... except Exception as inst:  
...     print type(inst)  
...     print inst.args  
...     print inst  
...     x, y = inst.args  
...     print 'x =', x  
...     print 'y =', y  
  
<type 'exceptions.Exception'>  
('spam', 'eggs')  
('spam', 'eggs')  
x = spam  
y = eggs
```





User defined exceptions

A user may define their own exceptions by deriving from the Exception class

```
class UserError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)

raise UserError('Oops')
```





User defined exceptions

Output

```
try:  
    raise UserError(2*2)  
except UserError as e:  
    print 'My exception occurred, value:', e.value  
  
My exception occurred, value: 4
```





Exception naming

Most exceptions are defined with names that end in “Error,” similar to the naming of the standard exceptions.





Common Exceptions

ZeroDivisionError

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo
by zero
```





Common Exceptions

NameError

```
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
```





Common Exceptions

TypeError

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int'
objects
```





Lab 8.2

Exceptions

PROBLEM

Define your own exception, then write a code block that raises it.

SUCCESS CRITERIA

Your program must use an advanced looping technique. It must also successfully raise your exception, and the exception must perform as expected.





Review Questions

-  How do you define your own Exception?
-  What type of Exception is a NameError?
-  What is the difference between Else and Finally?
-  What other data structure is a Dict similar to?



A decorative border of hexagonal icons surrounds the central content. The icons include a lightbulb, a thumbs-up, a network graph, a smartphone, a magnifying glass, a gear, a speech bubble, and a small teal hexagon at the bottom. A large, semi-transparent teal hexagon is positioned behind the number 9.

9

Classes





Unit Objectives

After this lesson, you will be able to:

-  Create classes in Python and instantiate their object
-  Explain Python's scoping rules and namespaces
-  Understand the inheritance model in Python





9.1 How classes work in Python





Classes – rules

- ◆ The class inheritance mechanism allows multiple base classes.
- ◆ A derived class can override any methods of its base class or classes
- ◆ A method can call the method of a base class with the same name





9.2 Private instance variables





Private Instance Variables

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python, although there is a convention that most Python users follow.





Private Instance Variables

A name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API, and be considered an implementation detail and subject to change without notice.

i.e. Please don't use it





9.3 Name mangling





Name mangling

Python believes that the use case for Class-Private Members, there is a mechanism called name-mangling available. If you use 2 leading underscores, `__spam` will be replaced by `_classname__spam`.





Name mangling

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update      # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```



9.4 Aliases





Aliases

Objects have individuality. Multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages.





Aliases

Aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation. If a function modifies an object passed as an argument, the caller will see the change.



9.5 Namespaces





Namespaces

A namespace is a mapping from names to objects. E.g. built-in exception names.

The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces.





Namespaces

According to Python, any name following a dot is called an attribute (e.g. function via module name). References to names in modules are attribute references.

```
modname . funcname
```





Namespaces

The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `_main_`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `_builtin_`.)





Namespaces

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function.



9.6 Scope





Scope

A scope is a textual region of a Python program where a namespace is directly accessible.





Scope

At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:





Available scopes

- ◇ The innermost scope, which is searched first, contains the local names
- ◇ The scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names





Available scopes

- ◆ The next-to-last scope contains the current module's global names
- ◆ The outermost scope (searched last) is the namespace containing built-in names





Available scopes

All variables found outside of the innermost scope are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).





9.6 Defining classes





Defining classes

The simplest form of class definition looks like this:

```
class ClassName:  
    <statement-1>  
    .  
    <statement-N>
```





Defining classes

Class definitions, like function definitions (def statements) must be executed before they have any effect.





9.7 Class objects





Class objects

When a class definition is left normally (via the end), a class object is created. This is basically a wrapper around the contents of the namespace created by the class definition.





Class objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python:
`obj.name`.





Class objects

```
class MyClass:  
    """A simple example class"""  
    i = 12345  
    def f(self):  
        return 'hello world'
```

MyClass.i and MyClass.f are valid attribute references, returning an integer and a function object.





Class objects

```
>>> MyClass.i = 999  
  
>>> MyClass.i  
999
```

Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment.





Class objects

DOCSTRINGS

`__doc__` is also a valid attribute, returning the docstring belonging to the class: "A simple example class".





Class objects

Class instantiation uses function notation.
Just pretend that the class object is a
parameterless function that returns a new
instance of the class. E.g.

```
x = MyClass()
```





`__init__`

Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):  
    self.data = []
```





`__init__`

When a class defines an `__init__()` method,
class instantiation automatically invokes
`__init__()` for the newly-created class
instance.





`__init__`

If the `__init__()` method has more arguments, then any arguments given to the class instantiation operator are passed on to `__init__()`.

```
class Complex:  
    def __init__(self, realpart, imagpart):  
        self.r = realpart  
        self.i = imagpart  
  
x = Complex(3.0, -4.5)  
print x.r, x.i  
3.0 -4.5
```





9.8 Instance objects





Instance objects

The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.





Instance objects

Data attributes correspond to instance variables in Smalltalk, and to data members in C++.





Instance objects

Data attributes need not be declared. Like local variables, they spring into existence when they are first assigned to.





Instance objects

The other kind of instance attribute reference is a method. A method is a function that “belongs to” an object.





9.9 Method Objects





Method objects

Usually, a method is called right after it is bound.

```
x.f()
```

However, it is not necessary to call a method right away. It can be stored away and called at a later time.

```
xf = x.f
while True:
    print xf()
```





Method objects

The special thing about methods is that the object is passed as the first argument of the function.

In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`.





9.10 General knowledge





Overwriting

Data attributes override method attributes with the same name, which may cause hard-to-find bugs in large programs. Best practice is to use some kind of convention that minimizes the chance of conflicts.





Naming conventions

Use verbs for methods and nouns for data attributes to avoid conflicts (or some other convention).





Data hiding

Classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention.





Self

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python.





Assigning a method

It is not necessary that the function definition is textually enclosed in the class definition.
Assigning a function object to a local variable in the class is also ok.





Assigning a method

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)
class C:
    f = f1
    def g(self):
        return 'hello world'
```





Methods calling methods

Methods may call other methods by using method attributes of the `self` argument

```
class Bag:  
    def __init__(self):  
        self.data = []  
    def add(self, x):  
        self.data.append(x)  
    def addtwice(self, x):  
        self.add(x)  
        self.add(x)
```



9.11 Bundling





Bundling

Sometimes it is useful to bundle together a few named data items. To do this we use an empty class definition.





Bundling

```
class Employee:  
    pass  
  
john = Employee() # Create an empty employee  
record  
  
# Fill the fields of the record  
john.name = 'John Doe'  
john.dept = 'computer lab'  
john.salary = 1000
```





Review Questions

- 🐾 Can data attributes overwrite method attributes?
- 🐾 What is usually the first argument of a method?
- 🐾 How do you automatically initialize variables on instantiation of a class?





10

Inheritance





Unit Objectives

After this lesson, you will be able to:



Explain how inheritance works in Python



Define multiple inheritance



10.1 Syntax





Syntax

The syntax for a derived class definition looks like this:

```
class DerivedClassName (BaseClassName) :  
    <statement-1>  
    .  
    .  
    <statement-N>
```





Scope

The name `BaseClassName` must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName (modname.BaseClassName) :
```





10.2 How it works





Procession

Execution of a derived class definition
proceeds the same as for a base class.
When the class object is constructed, the
base class is remembered.





Overriding

Derived classes may override methods of their base classes.





Calling the base class directly

There is a simple way to call the base class
method directly:

```
BaseClassName.methodname(self, arguments)
```





Extending instead of overriding

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name.





10.3 Inheritance built-ins





Built-in inheritance functions

Use `isinstance()` to check an instance's type:
`isinstance(obj, int)` will be True only if `obj.
__class__` is `int` or some class derived from
`int`.





Built-in inheritance functions

Use `issubclass()` to check class inheritance:
`issubclass(bool, int)` is True since `bool` is a subclass of `int`. However, `issubclass(unicode, str)` is False since `unicode` is not a subclass of `str` (they only share a common ancestor, `basestring`).





10.4 Multiple inheritance





Syntax

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
Class DerivedClassName(Base1, Base2, Base3):  
    <statement-1>  
    .  
    .  
    <statement-N>
```





New style vs old style

A "New Class" is the recommended way to create a class in modern Python.

A "Classic Class" or "old-style class" is a class as it existed in Python 2.1 and before. They have been retained for backwards compatibility.





Old style classes

For old-style classes, the only rule is depth-first, left-to-right.

Therefore, if an attribute is not found in DerivedClassName, it is searched in Base1, then (recursively) in the base classes of Base1, and only if it is not found there, it is searched in Base2, and so on.





New style classes

For new-style classes, the method resolution order changes dynamically to support cooperative calls to super().





New style classes

With new-style classes, dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottom-most class).





New style classes

For example, all new-style classes inherit from object, so any case of multiple inheritance provides more than one path to reach object.





The Dynamic Algorithm

To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents).





11

Modules





Unit Objectives

After this lesson, you will be able to:



Import modules into your code



Create packages in your application



Understand the difference between the various import commands





11.1 Importing modules





Importing modules

Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module. Definitions from a module can be imported into other modules or into the main module.





Definition

A module is a file containing Python definitions and statements suffixed with .py

```
import module
```





`__name__`

Within a module, the module's name (as a string) is available as the value of the global variable `__name__`.

```
>>> print __name__
__main__
```



11.2 Importing





Importing

Importing does not enter the names of the functions defined in file directly in the current symbol table. It only enters the module name file there. Using the module name you can access the functions. E.g.

```
import file  
file.add_nums(10)
```





Importing

To get the name of the file:

```
>>> file.__name__  
'file'
```





Importing

You can assign a function to a local name to make your code more readable.

```
func = file.add_nums  
func(100)
```





Modules – global variables

You can touch a module's global variables
with the same notation as above:

```
file.somevar
```





Modules – direct import

There is a variant of the import statement that imports names from a module directly into the importing module's symbol table.

```
>>> from file import add_nums, sub_nums  
>>> add_nums()
```





Modules - direct import

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, file is not defined).





Modules – import *

There is even a variant to import all names that a module defines. This imports all names except those beginning with an underscore _.

```
>>> from file import *
```





Modules - import *

Note that in general the practice of importing * from a module or package is frowned upon, since it often less readable code. However, it is okay to use it to save typing in interactive sessions.





11.4 Executing Modules as Scripts





Executing

You can execute a module. The module will execute normally, but with the `_name_` set to `_main_`.

```
python fibo.py <arguments>
```





Executing

If you want the module to act both as a main script, as well a module that is imported into other programs, you can use:

```
if __name__ == "__main__":
    import crab
    add_num(crab.claw("pincer"))
```



11.5 Dir





Dir

The built-in function `dir()` is used to find out which names a module defines.

```
>>> dir(fibo)
['__name__', 'fib', 'fib2']
```





Dir

Without arguments, `dir()` lists the names you have defined currently.

```
>>> dir(fibo)
['__name__', 'fib', 'fib2']
```





Dir

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
```





11.6 Packages





Packages

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name A.B designates a submodule named B in a package named A.





Packages

Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other's module names.





Packages

Suppose you want to design a collection of modules (i.e. a package) called sound.

```
sound/                                #Top-level package
    __init__.py                         #Initialize the sound package
    formats/                            #Subpackage for file format
        conversions
            __init__.py
            wavread.py
            wavwrite.py

    effects/                            #Subpackage for sound effects
        __init__.py
        echo.py
        surround.py
```





Packages – init

The `__init__.py` files are required to make Python treat the directories as containing packages.





Packages – init

This is done to prevent directories with a common name, such as String, from unintentionally hiding valid modules that occur later on the module search path.





Packages – init

In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.





11.7 Importing packages





Importing

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```





Importing

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.  
    7, atten=4)
```





Importing

An alternative way of importing the submodule is:

```
from sound.effects import echo
```





Importing

This also loads the submodule echo, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```





Importing

Yet another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```





Importing

Again, this loads the submodule echo, but this makes its function echofilter() directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```





Importing

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable.





11.7 Importing * from a package





Importing *

The import statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered.





Importing *

For example, the file sounds/effects/__init__.py could contain the following code:

```
__all__ = ["echo", "surround", "reverse"]
```





Importing *

This would mean that from `sound.effects` `import *` would import the three named submodules of the `sound` package.

```
__all__ = ["echo", "surround", "reverse"]
```





Importing *

If `__all__` is not defined, the statement from `sound.effects import *` may not import all submodules from the package `sound.effects` into the current namespace





Importing *

It only ensures that the package sound.effects has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package.





Importing *

This includes any names defined (and submodules explicitly loaded) by `__init__.py`.





Review Questions

- 🐾 How do we import everything from a package
- 🐾 Which import statements helps us avoid having to use the name of the module in each function call?
- 🐾 What do we need to place in a directory to tell Python that this is a package?





11

Developer modules

"My hovercraft is full of eels!"

Hungarian with translating book at the Tobacconist [spam]





Unit Objectives

After this lesson, you will be able to:



Use the JSON module



Read/write to a file



Work with other developer modules



11.1 JSON





JSON

JSON (JavaScript Object Notation) is a minimalistic format for structuring data. It is used primarily to transmit data between a server and web application, as an alternative to XML.





JSON

JSON is made up of key:value pairs, similar to a Dict in Python.

```
{  
    "collection" : {  
        "title" : "Blog",  
        "description" : "This is a description of my  
blog.",  
        "categories" : [ "Category-1", "Category-2" ]  
    }  
}
```





JSON

In fact, it's so similar that there's a module for converting JSON to Dictionaries, and conveniently, it's called the JSON module

Further reading:

<https://docs.python.org/2/library/json.html>





import json

To use the JSON module, you need to import it, and then you need to use its fully qualified name every time you call it, e.g. `json.loads()`.

```
import json  
  
json.dumps(a)
```





json.dumps

The `json.dumps` function takes a Python data structure and returns it as a JSON string.

```
a = {"name": "ben", "age": 35, "city": "Denver"}  
  
json.dumps(a)  
'{"city": "Denver", "age": 35, "name": "ben"}'
```





json.loads

The `json.loads()` function takes a JSON string and returns it as a Python Dict, which you can then go ahead and use.

```
a = '{"city": "Denver", "age": 35, "name": "ben"}'  
  
b = json.loads(a)  
{u'city': u'Denver', u'age': 35, u'name': u'ben'}  
  
b['city']  
u'Denver'
```



11.2 Read/Write to file





file reading/writing

There's no need to import any module. You can just start reading/writing to your file.





file reading/writing

`open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

```
>>> f = open('workfile', 'w')
>>> print f
<open file 'workfile', mode 'w' at 80a0960>
```





arguments

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used.





mode

read is the default, if nothing is specified

'r'	read only
'w'	only write
'a'	append to end
'r+'	read and write

```
f = open('workfile', 'w')
```





reading

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string. If size is not specified, it will read the entire file.

```
f = open('workfile', 'r')  
  
f.read()
```





reading a single line

f.readline() reads a single line from the file, and will be delimited by \n.

```
f = open('workfile', 'r')  
  
>>> f.readline()  
'This is the first line of the file.\n'  
  
>>> f.readline()  
'Second line of the file\n'
```





looping through a file

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
f = open('workfile', 'r')  
  
>>> for line in f:  
    print line,
```

This is the first line of the file.
Second line of the file





writing to a file

`f.write(string)` writes the contents of `string` to the file, returning `None`.

```
f = open('workfile', 'w')  
  
f.write('This is a test\n')
```





writing non-strings

To write something other than a string, it needs to be converted to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```





with

It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent `try-finally` blocks:

```
>>> with open('workfile', 'r') as f:  
...     read_data = f.read()  
>>> f.closed  
True
```





Additional File I/O Example





Unit Objectives

After this lesson, you will learn



basic file I/O syntax

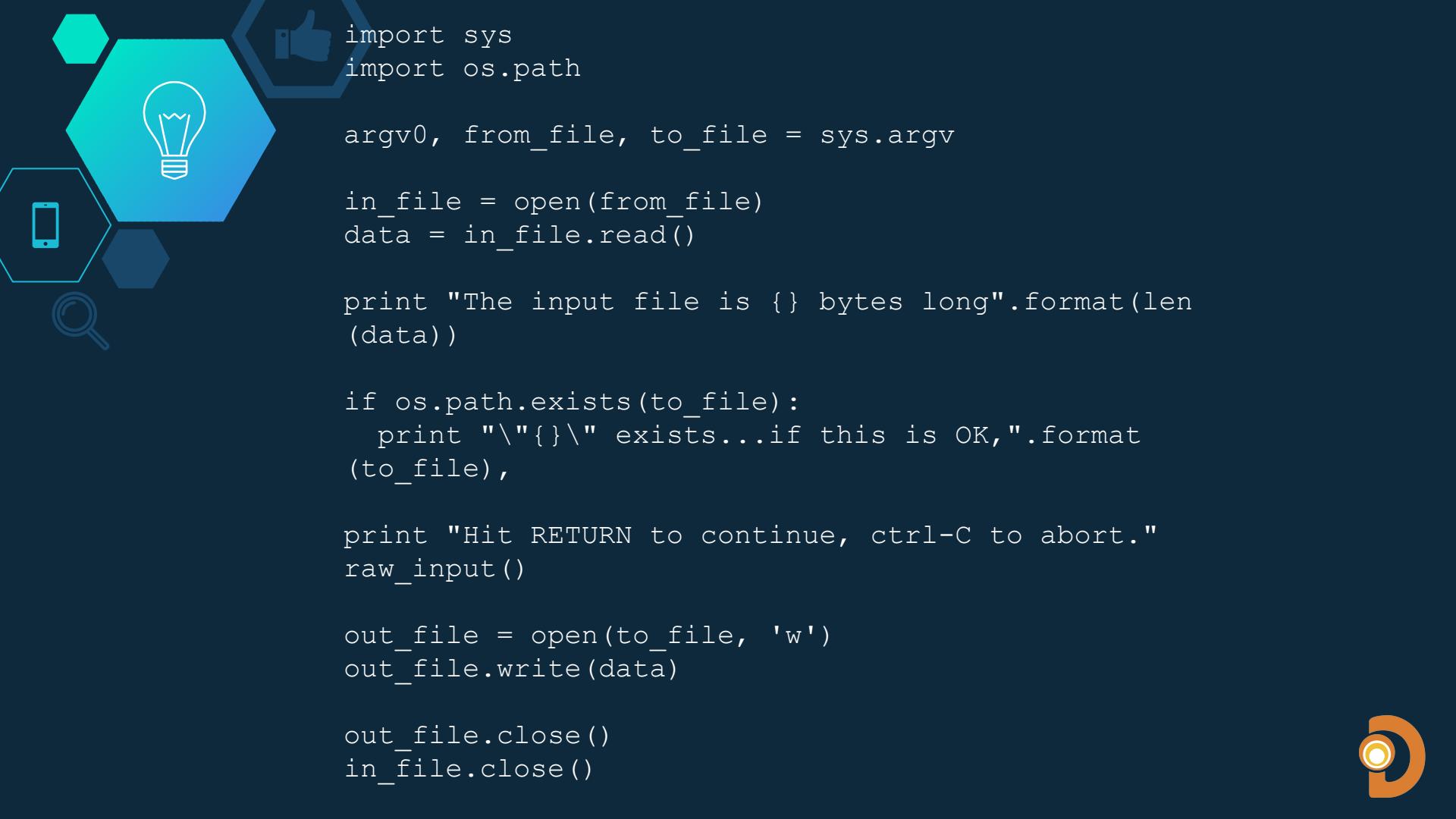


`raw_input()`



`import` vs. `from ... import`





```
import sys
import os.path

argv0, from_file, to_file = sys.argv

in_file = open(from_file)
data = in_file.read()

print "The input file is {} bytes long".format(len(data))

if os.path.exists(to_file):
    print "{} exists...if this is OK,".format(to_file),

print "Hit RETURN to continue, ctrl-C to abort."
raw_input()

out_file = open(to_file, 'w')
out_file.write(data)

out_file.close()
in_file.close()
```



```
from sys import argv      # import sys
from os.path import exists # import os.path

# argv0, from_file, to_file = sys.argv
argv0, from_file, to_file = argv

...
if exists(to_file):
    print "{} exists...if this is OK.".format(to_file),
```

- You could say that `from` ‘pollutes’ the namespace. Generally not a problem, but can be.
- If you use `from` to import multiple names which are the same (e.g., `os.path.exists`, and, say, `foo.exists`), the last import will “win”, i.e., ‘`exists`’ will refer to `foo.exists` not `os.path.exists`





raw_input() can be used to read a values to be stored for later use...

```
>>> name = raw_input("What is your name? ")  
What is your name? Dave  
>>> name  
'Dave'
```



Lab

File I/O

PROBLEM

Modify the preceding file I/O example to back up an existing destination file before overwriting it. The backup should be created in a unique temporary file, and the program should print out this temporary filename.



```
import sys
import os.path
import tempfile

argv0, from_file, to_file = sys.argv

if os.path.exists(to_file):
    in_file = open(to_file)
    data = in_file.read()
    temp = tempfile.NamedTemporaryFile(delete = False)
    temp.write(data)
    temp.close
    in_file.close()
    print "{} backed up in {}".format(to_file, temp.name)

in_file = open(from_file)
data = in_file.read()

out_file = open(to_file, 'w')
out_file.write(data)

out_file.close()
in_file.close()
```





Lab 11.2

The Big Lab

PROBLEM

Input must be collected from a User and stored in a file. The data should also be readable from the file by the app

SUCCESS CRITERIA

User should be offered a menu on whether to read or write data. Data must be stored as a dict, and when reading, App should print out only the values.





The Big Lab

- ◆ App needs to offer a User a menu.
- ◆ User should be able to choose “Create Product”, “List Products”, or “Exit Program”.
- ◆ If User chooses “Create Product”, they should be prompted for data. E.g New product name, new product price, etc.
- ◆ The product information the User inputted should be stored as a Dictionary, formatted as JSON and written to a local file.
- ◆ If User chooses “Product List”, App should read the contents of the JSON file, but only print the values to the screen





The Big Lab

- ◇ Product will need to have its own class
- ◇ You will need to write to a file called data.json
- ◇ Your main file should be called main.py
- ◇ Main.py should import Product
- ◇ You will need the JSON module for this task





Brief Intro to Decorators

Sometimes you want to modify a function's behavior without explicitly modifying the function, e.g., pre/post actions, debugging, etc.

To scratch the surface of decorators we need to look at some new concepts

- nested functions
- *args, **kwargs
- everything in Python is an object, even functions





```
def document_it(func):\n    def new_function(*args, **kwargs):\n        print('Running function:', func.__name__)\n        print('Positional arguments:', args)\n        print('Keyword arguments:', kwargs)\n        result = func(*args, **kwargs)\n        print('Result:', result)\n        return result\n\nreturn new_function
```





```
def add_ints(a, b):  
    return a + b
```

```
>>> print add_ints(3, 5)  
8
```

```
cooler_add_ints = document_it(add_ints)
```

```
>>> cooler_add_ints(3, 5)
```

Running function: add_ints

Positional arguments: (3, 5)

Keyword arguments: {}

Result: 8





```
@document_it  
def add_ints(a, b):  
    return a + b
```

```
>>> print add_ints(3, 5)  
Running function: add_ints  
Positional arguments: (3, 5)  
Keyword arguments: {}  
Result: 8  
8
```





Lab Review

In this lab, we used many of the different tools we learned during the course. Mostly, we learned how real-world applications transmit data, although usually, they would send HTTP requests to a RESTful API.





Review Questions

-  How do you safely open a file?
-  Does JSON need to be imported?
-  What does json.dumps do?
-  How do you loop through a file?



11.3 NetAddr





What it does

- IPv4 and IPv6 addresses, subnets, masks, prefixes
- iterating, slicing, sorting, summarizing and classifying IP networks
- dealing with various ranges formats (CIDR, arbitrary ranges and globs, nmap)
- set based operations (unions, intersections etc) over IP addresses and subnets
- parsing a large variety of different formats and notations
- looking up IANA IP block information
- generating DNS reverse lookups
- supernetting and subnetting





installing NetAddr

Use PIP to install, then you can import NetAddr, even from the interactive shell.

```
$ pip install netaddr
```





import NetAddr

You need to import netaddr using the from * notation. You also need to import pprint, which NetAddr uses for formatting

```
>>> from netaddr import *
>>> import pprint
```





Using NetAddr

Creating an IPAddress will instantiate and IPAddress object

```
>>> ip = IPAddress('192.0.2.1')
>>> ip.version
4
```





Using NetAddr

IPNetwork objects are used to represent subnets, networks or VLANs that accept CIDR prefixes and netmasks.

```
>>> ip = IPNetwork('192.0.2.1')
>>> ip.ip
IPAddress('192.0.2.1')

>>> ip.network, ip.broadcast
(IPAddress('192.0.2.1'), None)
```





CIDR NetAddr

There is also a property that lets you access the true CIDR address which removes all host bits from the network address based on the CIDR subnet prefix.

```
>>> ip.cidr  
IPNetwork('192.0.2.0/23')
```





Using NetAddr

Accessing an IP object using the `list()` context returns a list of all IP objects in the range specified by the IP object's subnet.

```
>>> ip = IPNetwork('192.0.2.16/29')
>>> ip_list = list(ip)
>>> len(ip_list)
8
>>> ip_list
[IPAddress('192.0.2.16'), IPAddress('192.0.2.17'), ...,
 IPAddress('192.0.2.22'), IPAddress('192.0.2.23')]
```





Thank you for
attending Core
Python!

