LAB 1 Introduction to Python – I

The purpose of this lab is to familiarize you with this term's lab system and to diagnose your programming ability and facility with Python. This course uses Python for all of its labs, and you will be called on to understand the functioning of large systems, as well as to write significant pieces of code yourself. While coding is not, in itself, a focus of this class, artificial intelligence is a hard subject full of subtleties. As such, it is important that you be able to focus on the problems you are solving, rather than the mechanical code necessary to implement the solution.

Python resources

Udacity offers a really helpful Introductory course for Python Beginners
https://classroom.udacity.com/courses/ud1110

Python: There are a number of versions of Python available. This course will use standard Python from http://www.python.org/. If you are running Python on your own computer, you should download and install latest version (Currently Python 3.7.4) from http://www.python.org/download/ .

1.1 Essentials of a Python program:

In most of today's written languages, words by themselves do not make sense unless they are in certain order and surrounded by correct punctuation symbols. This is also the case with the Python programming language. The Python interpreter is able to interpret and run correctly structured Python programs. For example, the following Python code is correctly structured and will run:

```
print("Hello, world!")
```

Many other languages require a lot more structure in their simplest programs, but in Python this single line, which prints a short message, is sufficient. A very informative example of Python's syntax which does (almost) exactly the same thing:

```
# Here is the main function.
def my_function():
    print("Hello, World!")

my_function()
```

A hash (#) denotes the start of a comment. The interpreter will ignore everything that follows the hash until the end of the line.

1.1.1 Flow of control

In Python, statements are written as a list, in the way that a person would write a list of things to do. The computer starts off by following the first instruction, then the next, in the order that they appear in the program. It only stops executing the program after the last instruction is completed. We refer to the order

in which the computer executes instructions as the flow of control. When the computer is executing a particular instruction, we can say that control is at that instruction.

1.1.2 Indentation and (lack of) semicolons

Many languages arrange code into blocks using curly braces **`{`** and **`}`** or `BEGIN` and `END` statements – these languages encourage us to indent blocks to make code easier to read, but indentation is not compulsory. Python uses indentation only to delimit blocks, so we must indent our code:

```
# this function definition starts a new block
def add_numbers(a, b):
    # this instruction is inside the block, because it's indented
    c = a + b
    # so is this one
    return c

# this if statement starts a new block
if it_is_tuesday:
    # this is inside the block
    print("It's Tuesday!")
# this is outside the block!
print("Print this no matter what.")
```

In many languages we need to use a special character to mark the end of each instruction – usually a semicolon. Python uses ends of lines to determine where instructions end (except in some special cases when the last symbol on the line lets Python know that the instruction will span multiple lines). We may optionally use semicolons – this is something we might want to do if we want to put more than one instruction on a line (but that is usually bad style):

```
# These all individual instructions -- no semicolons required!
print("Hello!")
print("Here's a new instruction")
a = 2

# This instruction spans more than one line
b = [1, 2, 3,
    4, 5, 6]

# This is legal, but we shouldn't do it
c = 1; d = 5
```

1.1.3 Built-in types

There are many kinds of information that a computer can process, like numbers and characters. In Python, the kinds of information the language is able to handle are known as types. Many common types are built into Python – for example integers, floating-point numbers and strings. Users can also define their own types using classes. In many languages a distinction is made between built-in types (which are often called "primitive types" for this reason) and classes, but in Python they are indistinguishable. Everything in Python is an object (i.e. an instance of some class) – that even includes lists and functions.

Python is a dynamically (and not statically) typed language. That means that we don't have to specify a type for a variable when we create it – we can use the same variable to store values of different types. However, Python is also strongly typed – at any given time, a variable has a definite type. If we try to

perform operations on variables which have incompatible types (for example, if we try to add a number to a string), Python will exit with a type error instead of trying to guess what we mean.

Python has large collection of built-in functions that operate on different kinds of data to produce all kinds of results. One function is called type, and it returns the type of any object.

```
>>> type(7)
<class 'int'>
>>>
```

1.1.3.1 Numbers:

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages.

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

1.1.3.2 Strings

A string is a sequence of characters. Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result. \ can be used to escape quotes. In the interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes.

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

Some common escape sequences:

Sequence	Meaning
\\	literal backslash
\'	single quote
\"	double quote
\n	newline
\t	tab

Sometimes we may need to define string literals which contain many backslashes – escaping all of them can be tedious. We can avoid this by using Python's raw string notation. By adding an **r** before the opening quote of the string, we indicate that the contents of the string are exactly what we have written, and that backslashes have no special meaning. For example:

```
# This string ends in a newline
"Hello!\n"

# This string ends in a backslash followed by an 'n'
r"Hello!\n"
```

Consider another example:

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'ununununium'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

1.1.4 Files

Although the `print` function prints to the console by default, we can also use it to write to a file. Here is a simple example:

```
with open('myfile.txt', 'w') as myfile:
    print("Hello!", file=myfile)
```

In the **`with`** statement the file **`myfile.txt`** is opened for writing and assigned to the variable **`myfile`**. Inside the **`with`** block, **`Hello!`** followed by a newline is written to the file. The **`w`** character passed to open indicates that the file should be opened for writing. The **`with`** statement automatically closes the file at the end of the block, even if an error occurs inside the block.

As an alternative to **`print`**, we can use a file's **`write`** method as follows:

```
with open('myfile.txt', 'w') as myfile:
    myfile.write("Hello!")
```

Unlike **`print`**, the **`write`** method does not add a newline to the string which is written.

We can read data from a file by opening it for reading and using the file's **`read`** method:

```
with open('myfile.txt', 'r') as myfile:
    data = myfile.read()
```

This reads the contents of the file into the variable data. Note that this time we have passed **`r`** to the open function. This indicates that the file should be opened for reading.

1.1.5 Variable scope and lifetime

Where a variable is accessible and how long it exists depend on how it is defined. We call the part of a program where a variable is accessible its scope, and the duration for which the variable exists its lifetime. A variable which is defined in the main body of a file is called a global variable. It will be visible throughout the file, and also inside any file which imports that file. Global variables can have unintended consequences because of their wide-ranging effects – that is why we should almost never use them. Only objects which are intended to be used globally, like functions and classes, should be put in the global namespace. A variable which is defined inside a function is local to that function. It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing.

Here is an example of variables in different scopes:

```
# This is a global variable
a = 0

if a == 0:
    # This is still a global variable
    b = 1

def my_function(c):
    # this is a local variable
    d = 3
    print(c)
    print(d)

# Now we call the function, passing the value 7 as the first and only parameter
my_function(7)

# a and b still exist
print(a)
print(b)

# c and d don't exist anymore -- these statements will give us name errors!
print(c)
print(d)
```

1.2 Selection control statements

1.2.1 Selection: **`if`** statement

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more **`elif`** parts, and the else part is optional. The keyword '**`elif`**' is short for '**`else if`**', and is useful to avoid excessive indentation. An if … elif … elif … sequence is a substitute for the switch or case statements found in other languages.

The interpreter will treat all the statements inside the indented block as one statement – it will process all the instruction in the block before moving on to the next instruction. This allows us to specify multiple instructions to be executed when the condition is met.

1.2.2 The **`for`** Statement

The **`for`** statement in Python differs a bit from what you may be used to in C. Rather than always giving the user the ability to define both the iteration step and halting condition (as C), 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. For example:

```
for i in range(1, 9):
    print(i)
```

Consider another example.

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

If you do need to iterate over a sequence of numbers, the built-in function **`range ()`** comes in handy. It generates arithmetic progressions:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

The given end point is never part of the generated sequence; range(10) generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the 'step'):

```
range(5, 10)
   5, 6, 7, 8, 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

1.3 Defining Functions

A function is a sequence of statements which performs some kind of task. Here is a definition of a simple function which takes no parameters and doesn't return any values:

```
def print_a_message():
    print("Hello, world!")
```

We use the **`def`** statement to indicate the start of a function definition. The next part of the definition is the function name, in this case `print_a_message`, followed by round brackets (the definitions of any

parameters that the function takes will go in between them) and a colon. Thereafter, everything that is indented by one level is the body of the function.

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

1.3.1 Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

This example also introduces the **`in`** keyword. This tests whether or not a sequence contains a certain value.

In Python, there can only be one function with a particular name defined in the scope – if you define another function with the same name, you will overwrite the first function. You must call this function with the correct number of parameters, otherwise you will get an error. Sometimes there is a good reason to want to have two versions of the same function with different sets of parameters. You can achieve something similar to this by making some parameters optional. To make a parameter optional, we need to supply a default value for it. Optional parameters must come after all the required parameters in the function definition:

```
def make_greeting(title, name, surname, formal=True):
    if formal:
        return "Hello, %s %s!" % (title, surname)

    return "Hello, %s!" % name

print(make_greeting("Mr", "John", "Smith"))
print(make_greeting("Mr", "John", "Smith", False))
```

When we call the function, we can leave the optional parameter out – if we do, the default value will be used. If we include the parameter, our value will override the default value.

1.3.2 Lambdas

We have already seen that when we want to use a number or a string in our program we can either write it as a literal in the place where we want to use it or use a variable that we have already defined in our code. For example, `print("Hello!")` prints the literal string `"Hello!"`, which we haven't stored in a variable anywhere, but `print(message)` prints whatever string is stored in the variable message.

We have also seen that we can store a function in a variable, just like any other object, by referring to it by its name (but not calling it). Is there such a thing as a function literal? Can we define a function on the fly when we want to pass it as a parameter or assign it to a variable, just like we did with the string "Hello!"?

The answer is yes, but only for very simple functions. We can use the `lambda` keyword to define anonymous, one-line functions inline in our code:

```
a = lambda: 3

# is the same as

def a():
    return 3
```

Lambdas can take parameters – they are written between the lambda keyword and the colon, without brackets. A lambda function may only contain a single expression, and the result of evaluating this expression is implicitly returned from the function (we don't use the `return` keyword):

```
b = lambda x, y: x + y

# is the same as

def b(x, y):
    return x + y
```

1.4 Class Definitions:

A class serves as the primary means for abstraction in object-oriented programming. In Python, every piece of data is represented as an instance of some class. A class provides a set of behaviors in the form of member functions (also known as methods), with implementations that are common to all instances of that class. A class also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of attributes (also known as fields, instance variables, or data members).

The simplest form of class definition looks like this:

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

Example: CreditCard Class

As a first example, we will define a `CreditCard` class. The instances defined by the `CreditCard` class provide a simple model for traditional credit cards. They have identifying information about the customer, bank, account number, credit limit, and current balance. The class restricts charges that would cause a card's balance to go over its spending limit, but it does not charge interest or late payments.

The construct begins with the keyword, class, followed by the name of the **`class`**, a colon, and then an indented block of code that serves as the body of the class. The body includes definitions for all methods of the class. These methods are defined as functions, yet with a special parameter, named **`self`**, that serves to identify the particular instance upon which a member is invoked.

By convention, names of members of a class (both data members and member functions) that start with a single underscore character (e.g., `_secret`) are assumed to be nonpublic and should not be relied upon.

1.4.1 The **`self`** Identifier:

In Python, the `self` identifier plays a key role. In the context of the `CreditCard` class, there can presumably be many different `CreditCard` instances, and each must maintain its own balance, its own credit limit, and so on. Therefore, each instance stores its own instance variables to reflect its current state.

Syntactically, `self` identifies the instance upon which a method is invoked. For example, assume that a user of our class has a variable, `my_card`, that identifies an instance of the `CreditCard` class. When the user calls `my_card.get_balance( )`, identifier self, within the definition of the `get_balance` method, refers to the card known as `my_card` by the caller. The expression, `self._balance` refers to an instance variable, named `_balance`, stored as part of that particular credit card's state.

```
class CreditCard:
  """A consumer credit card."""

  def __init__(self, customer, bank, acnt, limit):
    """Create a new credit card instance.

    The initial balance is zero.

    customer  the name of the customer (e.g., 'John Bowman')
    bank      the name of the bank (e.g., 'California Savings')
    acnt      the acount identifier (e.g., '5391 0375 9387 5309')
    limit     credit limit (measured in dollars)
    """
    self._customer = customer
    self._bank = bank
    self._account = acnt
    self._limit = limit
    self._balance = 0

  def get_customer(self):
    """Return name of the customer."""
    return self._customer

  def get_bank(self):
    """Return the bank's name."""
    return self._bank

  def get_account(self):
    """Return the card identifying number (typically stored as a string)."""
    return self._account

  def get_limit(self):
    """Return current credit limit."""
    return self._limit

  def get_balance(self):
    """Return current balance."""
    return self._balance
```

Code Fragment 2.1: The beginning of the CreditCard class definition (continued in Code Fragment 2.2).

```
  def charge(self, price):
    """Charge given price to the card, assuming sufficient credit limit.

    Return True if charge was processed; False if charge was denied.
    """
    if price + self._balance > self._limit:      # if charge would exceed limit,
      return False                               # cannot accept charge
    else:
      self._balance += price
      return True

  def make_payment(self, amount):
    """Process customer payment that reduces balance."""
    self._balance -= amount
```

Code Fragment 2.2: The conclusion of the CreditCard class definition (continued from Code Fragment 2.1). These methods are indented within the class definition.

1.4.2 The Constructor

A user can create an instance of the `CreditCard` class using a syntax as:

```
cc = CreditCard( John Doe, 1st Bank , 5391 0375 9387 5309 , 1000)
```

Internally, this results in a call to the specially named `__init__` method that serves as the constructor of the class. Its primary responsibility is to establish the state of a newly created credit card object with appropriate instance variables. By the conventions, a single leading underscore in the name of a data member, such as `_balance`, implies that it is intended as nonpublic. Users of a class should not directly access such members.

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 = []
```

Testing the Class:

We will demonstrate some basic usage of the `CreditCard` class, inserting three cards into a list named `wallet`. We use loops to make some charges and payments, and use various accessors to print results to the console. These tests are enclosed within a conditional, `if __name__ == '__main__':`, so that they can be embedded in the source code with the class definition.

```
53 if __name__ == '__main__':
54   wallet = [ ]
55   wallet.append(CreditCard('John Bowman', 'California Savings',
56                            '5391 0375 9387 5309', 2500) )
57   wallet.append(CreditCard('John Bowman', 'California Federal',
58                            '3485 0399 3395 1954', 3500) )
59   wallet.append(CreditCard('John Bowman', 'California Finance',
60                            '5391 0375 9387 5309', 5000) )
61
62   for val in range(1, 17):
63     wallet[0].charge(val)
64     wallet[1].charge(2*val)
65     wallet[2].charge(3*val)
66
67   for c in range(3):
68     print('Customer =', wallet[c].get_customer())
69     print('Bank =', wallet[c].get_bank())
70     print('Account =', wallet[c].get_account())
71     print('Limit =', wallet[c].get_limit())
72     print('Balance =', wallet[c].get_balance())
73     while wallet[c].get_balance( ) > 100:
74       wallet[c].make_payment(100)
75       print('New balance =', wallet[c].get_balance())
76     print()
```

Code Fragment 2.3: Testing the CreditCard class.

1.4.3 Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```