



CSE4IP

Lecture 2 - Hello world, variables, numbers and assignments

Ayad Turkey

Last time

- Computers store data in the form of bits (0 or 1) and bytes (sets of 8 bits)
- For the most part, computers do exactly what a programmer tells them
- Computers only understand a special language of 1s and 0s
 - We call this *machine language* or *machine code*
- One level up from machine language is *assembly language*
 - Still dependent on the specific hardware
- Almost all actual programming is done in *high-level languages*

So, we're going to write programs in Python and run them

Let's take our first look at what that might look like, shall we?

Hello, world

hello_world.py

```
# Display "Hello, World!" on the screen  
print("Hello, World!")
```

Our first Python program

Hello, world

hello_world.py

```
# Display "Hello, World!" on the  
screen
```

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

Output

Hello, world

The following slides explain what this program is telling the computer. However, note that we don't expect you to fully understand this program right now.

Future lectures and lab sessions will focus specifically on different aspects of this program.

Hello, world

hello_world.py

```
# Display "Hello, World!" on the  
screen
```

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

The hash sign/pound sign (**#**) at the beginning of a line in Python has a special meaning – it indicates that anything on a line after the hash character (**#**) will be ignored by the Python interpreter.

Hello, world

hello_world.py

```
# Display "Hello, World!" on the  
screen
```

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

The second line is a statement. It asks the interpreter to display the message “Hello, World!” on the screen.

print is a built-in function to display information on the screen. A built-in function is one that is pre-defined and is always available for use.

Hello, world

hello_world.py

```
# Display "Hello, World!" on the  
screen  
print("Hello, World!")
```

Now, suppose we type this program into a **text file** called **hello_world.py**. Then we can run it by entering the following command at the operating system command prompt:

python hello_world.py

We should see Hello, World! displayed on the screen.

Running a program in this way is known as “working with Python in script mode”.

Combining the two modes

- We can combine the two modes by using IDLE, which is the editor that comes with Python software.
- When we run a program in IDLE, by choosing menu option Run > Run Module, after the execution of the program
 - An interactive shell will be available for further interaction.
 - In the subsequent interaction, everything we have defined in the program are available for us to use.

Combining the two modes

- As an example, consider the following program:

p1_simple_add.py

```
# Assign two values to n1 and n2
n1 = 10
n2 = 20

# Add the numbers
total = n1 + n2

# Display the result
print("The total is ", total)
```

When we run program in IDLE (by choosing the Run > Run Module option), three things happen:

1. An interactive shell is opened
2. The program is executed and the output is displayed in the shell window, and
3. The shell prompt >>> is displayed, ready to respond to our requests

Combining the two modes

- As an example, consider the following program:

p1_simple_add.py

```
# Assign two values to n1 and n2

n1 = 10

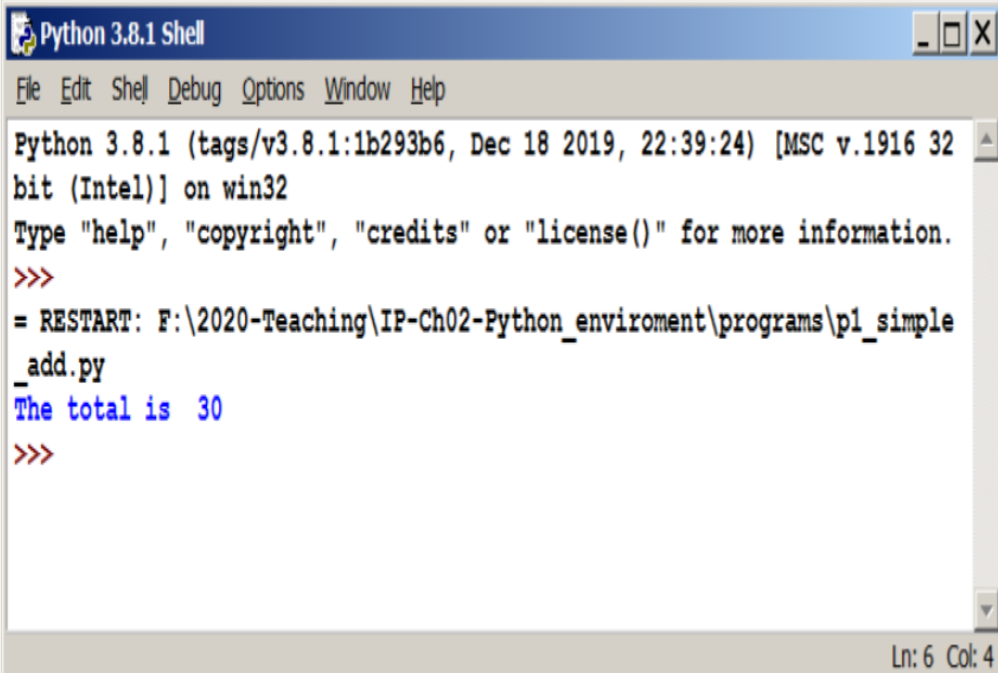
n2 = 20

# Add the numbers

total = n1 + n2

# Display the result

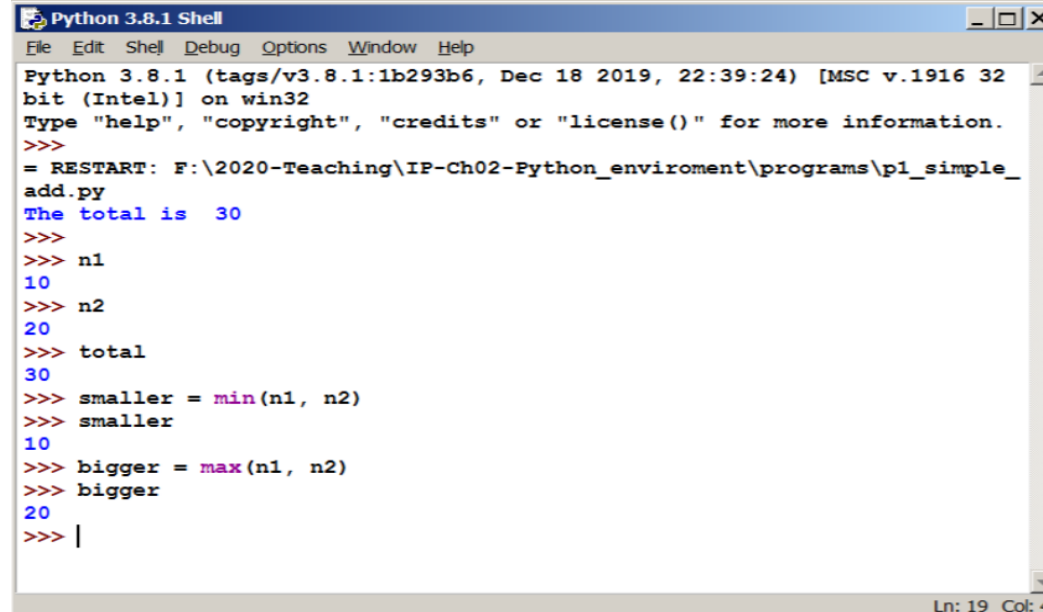
print("The total is ", total)
```



The screenshot shows a 'Python 3.8.1 Shell' window. The title bar is blue with the Python logo and text. The menu bar includes 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area shows the following output:
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: F:\2020-Teaching\IP-Ch02-Python_enviroment\programs\p1_simple_add.py
The total is 30
>>>
The status bar at the bottom right indicates 'Ln: 6 Col: 4'.

Combining the two modes

- While the shell is active, Python remembers all the definitions in the program: the definitions of variables `n1`, `n2` and `total`,
- And in the subsequent interaction, we can access them and use them as we wish. A sample interaction is shown below:



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32
bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: F:\2020-Teaching\IP-Ch02-Python_enviroment\programs\p1_simple_
add.py
The total is 30
>>>
>>> n1
10
>>> n2
20
>>> total
30
>>> smaller = min(n1, n2)
>>> smaller
10
>>> bigger = max(n1, n2)
>>> bigger
20
>>> |
```

Combining the two modes

- In this sample interaction, we
- Display $n1$, $n2$ and total
- Compute the minimum of $n1$ and $n2$ and display it
- Compute the maximum of $n1$ and $n2$ and display it
- We have used function **min**, which can take two more numbers and returns the minimum of these numbers. Similarly for function **max**.

More Examples

- In this lecture, we will have a look at some more examples.
- You will encounter some of the most commonly used programming elements, which will be studied in detail later.

Example 1- Personal Welcome

- In the program below, we get some input from the user:

welcome.py

```
# Get the name from the user
name = input("Please enter your name: ")
# Display the welcome message
print("Welcome", name, "!")
```

Suppose we type this program into a text file called welcome.py. Then we can run it with the command

python welcome.py

The program will prompt us for a name with the prompt message

Please enter your name:

Now, suppose we type in John, then we get the sample run shown below:

Please enter your name: John
Welcome John !

Example 1- Personal Welcome

- In the program below, we get some input from the user:

welcome.py

```
1 # Get the name from the user
2 name = input("Please enter your name: ")
3
4 # Display the welcome message
5 print("Welcome", name, "!")
```

Let us have a look at the program.

On line 2, we use the **input** function to prompt the user for a name, read it and assign it to variable name.

On line 5, the print function takes three items and displays them on the screen. The first item is a string, the second a variable, and the third a string. The items are displayed with a space between them.

Example 2- Adding Numbers

- In the previous example, the program gets an input from the user, and the input is actually a **string**. In this example, the program gets two **numbers** (integers, to be more specific).
- The program below, which asks the user for two numbers, adds them up and displays the total on the screen:

Example 2- Adding Numbers

```
1 # Get two integers entered by the user
2 n1 = int(input("Enter the first number: "))
3 n2 = int(input("Enter the second number: "))
4
5 # Add the numbers
6 total = n1 + n2
7
8 # Display the result
9 print("The total is ", total)
```

Sample run:

Enter the first number: 10

Enter the second number: 20

The total is 30

A Look at the Program

On line 2 (and line 3), the program uses two built-in functions **input** and **int** in combination.

The first function reads a string from the keyboard, and the second converts the input string into an integer (which is a value of type `int`).

Example 2- Adding Numbers

- We can see how they work together by performing the two steps separately, in sequence:

```
>>> s = input("Enter a number: ")
```

```
Enter a number: 10
```

```
>>> s
```

```
'10'
```

```
>>> type(s)
```

```
<class 'str'>
```

```
>>> n = int(s)
```

```
>>> s
```

```
'10'
```

```
>>> type(n)
```

```
<class 'int'>
```

Example 2- Adding Numbers

- As a further illustration about getting input from the user, the program below shows how we can get, as input, a string, an integer or a real number.
- A real number, which has the whole number part and the decimal part, is also known as a floating-point or simply a float.

```
1 s = input("Enter a string: ")
2 n = int(input("Enter an integer: "))
3 x = float(input("Enter a real number: "))
4 print()
5 print("Inputs received:", s, n, x)
```

Example 2- Adding Numbers

```
1 s = input("Enter a string: ")
2 n = int(input("Enter an integer: "))
3 x = float(input("Enter a real number: "))
4 print()
5 print("Inputs received:", s, n, x)
```

On line 3, function float takes a string and converts it into a real number (a float value).

The statement on line 4 displays a blank line.

Sample run:

Enter a string: Hello
Enter an integer: 12
Enter a real number: 34.56

Inputs received: Hello 12 34.56

Example 3- Area of a Circle

```
1 # This program computes the area of a circle. It receives the radius as
2 # an input from the user.
3
4 # Import the math library so that we can use constant pi.
5 import math
6
7 # Get the radius
8 radius = float(input("Enter the radius: "))
9
10 # Compute the area
11 area = math.pi * radius**2
12
13 # Display the area
14 print("The circle area of the circle is ", round(area, 2))
```

Running the program in IDLE, we can get the sample run as such the one shown below:

```
Enter the radius: 4.5
The circle area of the circle is 63.62
>>>
```

And we can follow up with a few requests

```
>>> radius
4.5
>>> area
63.61725123519331
>>> math.pi
3.141592653589793
```

Example 3- Area of a Circle

```

1 # This program computes the area of a circle. It receives the radius as
2 # an input from the user.
3
4 # Import the math library so that we can use constant pi.
5 import math
6
7 # Get the radius
8 radius = float(input("Enter the radius: "))
9
10 # Compute the area
11 area = math.pi * radius**2
12
13 # Display the area
14 print("The circle area of the circle is ", round(area, 2))

```

```

>>> area
63.61725123519331
>>> round(area, 2)
63.62
>>> area
63.61725123519331

```

math is a module (line 5). A module is a collection of statements in a file with the .py extension. Typically, these statements define a number of functions, classes, variables and constants. A module is also known as a library.

To use a module, we need to **import** it first (except for what are known as built-in modules)

math.pi is a constant defined in the math module.

round is a built-in function. In this particular case, the function takes the value of area and returns that value rounded to the nearest number with 2 decimal places. **It does not change the value of area**

Three Types of Program Errors

- There are three types of program errors: compile-time errors, run-time errors and logic errors:
 - **Compile-Time Errors:** Consider this program (add_numbers_v1.py), which is meant to add two numbers:

```
1 # This program adds two numbers
2 x = 10
3 total = x * y
4 print("the total is"; total)
```


Three Types of Program Errors

```
1 # This program adds two numbers
2 x = 10
3 total = x * y
4 print("the total is"; total)
```

- When we run the above program from the command-line, we get an error message

```
C:>python add_numbers_v1.py
File "add_numbers_1.py", line 4
    print("The total is"; total)
                        ^
SyntaxError: invalid syntax
```

What we have here is a syntax error. **On line 4**, we should have a comma instead of a semi-colon.

Syntax errors are also known as compile-time errors.

Three Types of Program Errors

- **Run-Time Errors:** Let us fix the program and rename it to `add_numbers_v2.py`

```
1 # This program adds two numbers
2 x = 10
3 total = x * y
4 print("The total is", total)
```

Now, when we run the program, we get the error message:

```
C:>python add_numbers_v2.py
Traceback (most recent call last):
  File "add_numbers_v2.py", line 3, in <module>
    total = x + y
NameError: name 'y' is not defined
```

The problem here is that when the statement **on line 3** is executed, an illegal operation is encountered: we have tried to use variable **y**, but **y** is undefined. Errors of illegal operations are known as run-time errors.

Three Types of Program Errors

- **Run-Time Errors:** Let us fix the program and rename it to `add_numbers_v2.py`

```
1 # This program adds two numbers
2 x = 10
3 total = x * y
4 print("The total is", total)
```

Now, when we run the program, we get the error message:

```
C:>python add_numbers_v2.py
Traceback (most recent call last):
  File "add_numbers_v2.py", line 3, in <module>
    total = x + y
NameError: name 'y' is not defined
```

The problem here is that when the statement **on line 3** is executed, an illegal operation is encountered: we have tried to use variable **y**, but **y** is undefined. Errors of illegal operations are known as run-time errors.

Three Types of Program Errors

- Why we have the terms “compile-time error” and “run-time error”?
- When we run a Python program, this is done in two steps:
 - **Step 1 - Compiling the source code:** The statements in our text file are translated into what is known as the byte code.
 - **Step 2 - Executing the byte code:** The statements in byte code are executed, one at a time. Syntax errors are detected in the first step, and illegal operation errors are detected in the second steps. Hence the names

Note that when we run a program in IDLE (instead of in script mode), the way the errors are presented make it is

Three Types of Program Errors

- **Logic errors:** Now suppose we fix the run-time error and get the following program add_numbers_v3.py:

```
1 # This program add two numbers
2 x = 10
3 y = 20
4 total = x * y
5 print("The total is", total)
```

- Run the program, we get

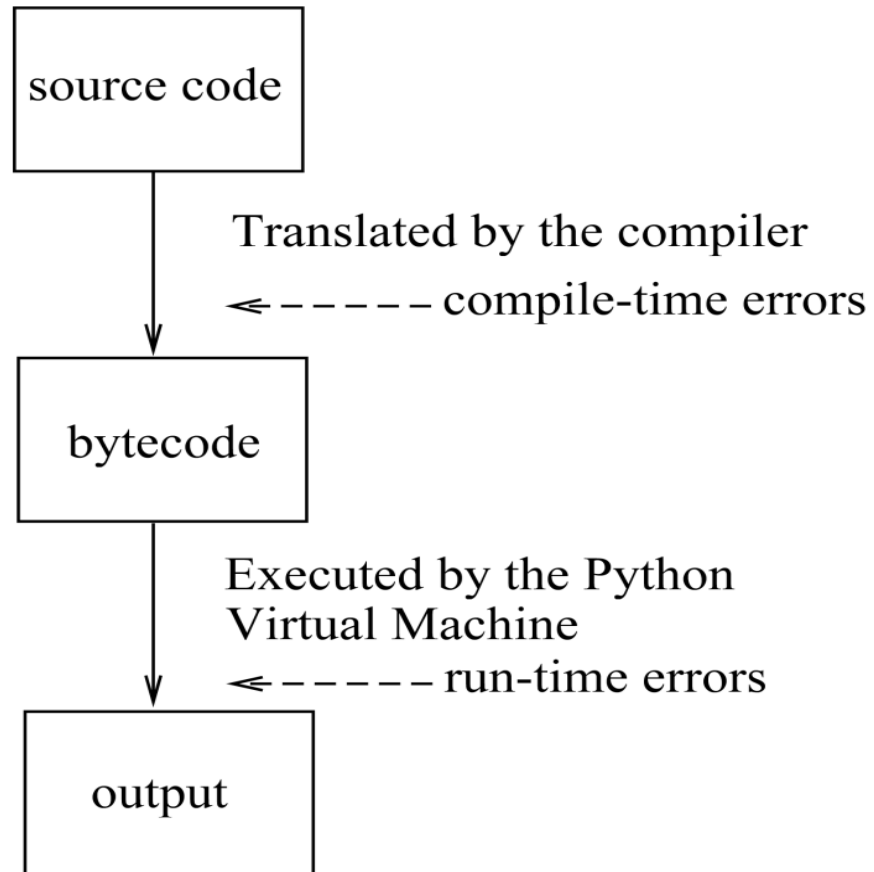
```
C:>python add_numbers_v3.py
The total is 200
```

The program runs and gives us an output. But the result is incorrect: it is the product of the two numbers, not their sum. We have given the computer the wrong instruction. This is a logic error. (Of course, we know how to fix this particular error.)

Three Types of Program Errors

- Compile-time error = Syntax error
- Run-time error = Illegal operation
- Logic error = Wrong instruction

Python/run process



Get help

- List things in a module

`dir(module)`

- List functions we can perform on an object

`dir(object)`

- Get help information about a function

`help(function)`

Get help

- List things in a module

`dir(module)`

- List functions we can perform on an object

`dir(object)`

- Get help information about a function

`help(function)`

Variables

- Most computer programs perform the following tasks:

Receiving input -- Processing data -- Producing output

- Input is any data that the program receives during the execution of the program. Once the input is received, some processing is normally performed on it. The result of this processing is then sent out to the environment as output
- In this process, the program stores data in the computer's memory and uses variables as the means to access and make use of the data.
- A variable is a name that refers to a value stored in the computer's memory.

Defining Variables and

Assignment Statement

We define a variable by the assignment statement, which has the general syntax:

<variable> = <expression>

- A variable can be assigned more than once to different values, even to values of different types.
- A variable, by itself, does not have a type. It takes the type of the value that it refers to.

Defining Variables and Assignment Statement

```
>>> x = 10                # assign x to 10
>>> x
10
>>> type(x)
<class 'int'>

>>> x = 20                # assign x to 20
>>> x
20
>>> type(x)
<class 'int'>

>>> x = "Hello there!"    # assign x to a string
>>> x
'Hello there!'
>>> type(x)
<class 'str'>
```

- We must define a variable before using it. Otherwise, we will have a run-time error

```
>>> y * 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

Rules for Variable Names

- A variable name must
 - begin with a letter or the underscore character,
 - followed by zero or more letters, digits and underscores
- Variable names are case-sensitive, i.e. uppercase and lowercase characters are treated as being distinct.
- A variable name must not be one of the keywords.

Rules for Variable Names

- The list of keywords is shown below:

```
>>> help("keywords")
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.
```

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

Choosing Variable Names

- We must carefully choose the names for variables to reflect what they stand for.
- Underscore and camel case conventions
- Often to reflect what it stands for, we may use a variable name that consists of *more than one words*. In such as case, there are two common conventions that we can follow:
 - Use the underscore character to represent a space, e.g.
`pay_rate`, `gross_pay`, `number_of_computers_per_lab`

Choosing Variable Names

- Use the camel case convention in which the first character of the second and subsequent words is written in uppercase, e.g.

`payrate`, `grossPay`, `numberOfComputersPerLab`

Which convention should we use? The first convention is very popular among Python programmers. The names look good when standing alone. But in an expression with several of them, the expression can be quite hard to read. Hence, in this subject, we will generally adopt the second convention.

Avoiding using function names as variable names

We must also very careful and avoid using a built-in function name as a variable name. Otherwise, the name is no longer the name of the function and therefore cannot be used to call the function.

- The following example shows us the potential trans of this kind:

```
>>> aList = [1, 2, 3, 4]           # this is a list in Python

>>> sum(aList)                     # function 'sum' computes the sum of the list
's numbers'
10

>>> sum = 1 + 2                   # suppose we use 'sum' as a variable name

>>> sum(aList)                     # 'sum' no longer refer to the function
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    sum(aList)
TypeError: 'int' object is not callable

>>> __builtins__.sum(aList)        # if desperate, we can call it with the
module name
10
```

Multiple Assignments with a Single Statement

Python provides a variation of the basic assignment statement that allows us to assign several values to several variables in a single statement.

```
>>> x, y = 10, 20
>>> x
10
>>> y
20
```

- This feature can be very useful in some cases. Here is an example in which we interchange the values of two variables:

```
>>> x = 10
>>> y = 20
>>> x, y = y, x
>>> x
20
>>> y
10
```

Multiple Assignments with a Single Statement

Here is another example. Does the pattern of calculation seem familiar to you (some of you)?

```
>>> a = 1
>>> b = 1
>>> a, b = b, a+b
>>> b
2
>>> a, b = b, a+b
>>> b
3
>>> a, b = b, a+b
>>> b
5
```

Numbers and Strings: Some General Concepts

Numbers and strings are the basic data types manipulated by programs. Most other data types, e.g. list, are built on top of these.

- Integers are whole numbers, e.g. 123
- Real numbers are numbers that can have fractional parts.
- The fractional part of a real number is most often represented by the digits after the decimal point, e.g. 123.45.
- The fractional part is also referred to as the decimal parts, and the digits in the decimal part are referred to as decimal digits.
- Real numbers are also referred to as floating-point numbers or simply floats.

Why the term “floating-point”?

- To represent a wide range of values, using a limited amount of space in computer memory, real numbers are represented by what is known as the floating-point representation (or floating-point notation).
- What is this floating-point representation? Consider, for example, this number 0.000123. It can be written as 123×10^{-6} . In other words, it can be represented by two parts: the first part is 123, which is known as the **mantissa**, and the second part is -6 , which is known as the **exponent**.
- Now, consider this number 123000. It can be written as 123×10^3 . The mantissa is 123 (the same as for the previous number) and the exponent is 3.

Why the term “floating-point”?

- As we can infer from these two examples, the floating-point representation allows us to represent a wide range of values, from the very small ones to the very large ones. The actual decimal points for those numbers are not fixed at a position, but can ‘float around’, dependent on the value of the exponent. Hence the name.
- To use numbers and strings, we need to know the following basic things:
 - How to write numbers and strings in a program
 - How to manipulate them with operators and functions
 - How to perform input and output on them

Recap

- Python is a pretty good programming language
- In this subject, we will be writing programs in Python
- We went over hello.py and some other examples in gory detail
- We went over variables, numbers and assignments



NEXT LECTURE: Numbers, Arithmetic, Math functions and String



Thank
you.

Be
well.

latrobe.edu.au