KIT100 PROGRAMMING PREPARATION

Lecture Three:

Storing and modifying data

Lecture Objectives

- Storing Data
- Defining the essential Python data types
- Basic data inputs and outputs

Remember

- Computers are very quick and can perform complicated tasks
- Computers work to strict rules – they must be told what we need them to do exactly as they require it
- The CPU is the computers brain, it follows the fetch, decode, execute cycle
- Data is stored in a computer's memory to specific locations with unique identifiers

C.R.U.D

- Create, Read, Update, and Delete
  - Tasks that computers will perform on the data we want to create, access, modify or remove.
  - Our applications will always do one of the C.R.U.D tasks on the data we tell it to.

Programming for Problem Solving

- The purpose of writing a program is to **solve a problem**
  - Program as a model of part of the real world
- The general steps in problem solving are:
  1. *Understand* the problem (*don't solve the wrong problem*!)
  2. Dissect the problem into manageable pieces
  3. *Design* a solution
  4. Consider alternatives to the solution and refine it
  5. Implement the solution
  6. Test the solution and fix any problems that exist

Program Development

The creation of software involves four basic activities:

1. establish the **requirements**

   WHAT the program must do

2. create a **design** (HOW to solve the problem)

   The ALGORITHM (a sequence of statements)

3. **implement** the code

   Use INCREMENTAL development

4. **test** the implementation

   Incremental testing. The longer an error stays in a program the more expensive it is to fix

Algorithms

- How to solve the problem
  - What is a problem?

- known input; or
- physical location; or
- existing application; or
- …

- calculated result; or
- different location; or
- new function; or
- …

Example – **3.1 PP task:**
A car is traveling at a constant speed of 50 kilometres per hour.
We know '*distance / time = speed*'. Write a program that displays the following:

It will take the car 1.0 hour(s) to travel 50 kilometres
It will take the car 2.0 hour(s) to travel 100 kilometres

Algorithms

An **algorithm** is a set of instructions for performing a task:

	Sequence	• order is important; must be able to determine the next step
of	Steps	• each instruction must be clear and executable
that eventually	Stops	• the process must terminate
and which	Succeeds	• the goal must be achieved on termination

Storing Data

- Declaration and use of variables
- Primitive/raw data
- Assigning values to **variables**

What is a variable?

- A ***VARIABLE*** is just a name for a location in *memory*
- In many programming languages, a variable must be declared (before you use it), specifying
  - the variable's name (identifier)
  - the type of information that will be held in it (for example *number* type)

- Not in Python (good news)! Whenever you want to create a new variable, *just start using it*!

Declaring variables

- Python allows you to create a name for your variable, **without** first defining the type.
- Convenient? *You may make a mistake*
  - you must also remember to always use **unique** **variable** identifiers, otherwise you may mistakenly overwrite your old variable that has the same name.
  - More formally, ***you can only have one variable with the same name within the same scope***[1]

```
>>> name = "Jimmy"
>>> print(name)
Jimmy
>>> name = 5
>>> print(name)
5
```

- Spelling and correct use of **letter cases** is extremely important in Python! A more subtle danger is that if you create a variable that you intend to use more than once, and you spell it incorrectly in one of those uses – it's a different variable!

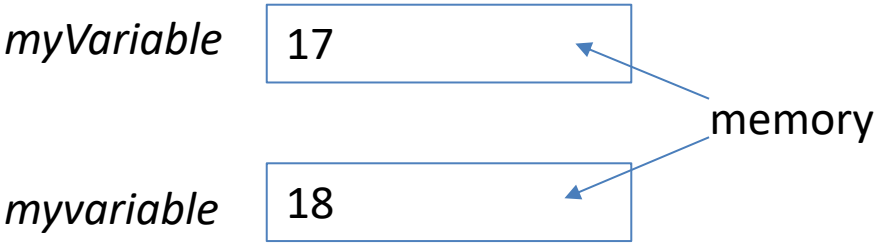
[1] *Think of scope to mean an area or region of your program*

Declaring variables

Uppercase

```
myVariable = 17
myvariable = 18
```

Lowercase



These are not the same variable!

Declaring variables – data types

- Four kinds of primitive data types exist in Python
  1. Whole numbers (`integers`) e.g. -2, 0, 1, 32768
  2. Floating point numbers (`reals` or `floats`) e.g. 1.34
  3. Sequences of (zero of more) characters (`string`) "hello"
  4. Boolean values – either `True`, or `False`

- Question: 1 equals to 1.0? -> False // 1 (int type); 1.0 (float type);

- In actual fact the value is **not** stored directly in the variable but rather the variable holds **the memory address** of where the value can be found. But conceptually we can just think of variables as boxes with values in them.

Integers

- Any whole number is an integer.
- 1 is an integer, 1.0 is not.
- An integer is represented by the `int` data type.

Reals

- Any number that has a decimal point is a floating point number, or a real number.
- 5.0 is a floating point number and 5 is not. Python stores floating point numbers in the **`float`** data type.

String

- String: sequence of characters used as data, the `str` data type
- String literal: a string that appears in actual code of a program
- Must be enclosed in single ', double " or triple-single ''' or triple-double """ quote marks (the same mark must be used at the beginning and the end of the string). Usually use double quotes "
  - An enclosed string can contain both single and double quotes and can have multiple lines

```
print("I'm here!")
print('I am reading "Hamlet" tonight')
print ('''I'm
reading "Hamlet"
tonight''')
```

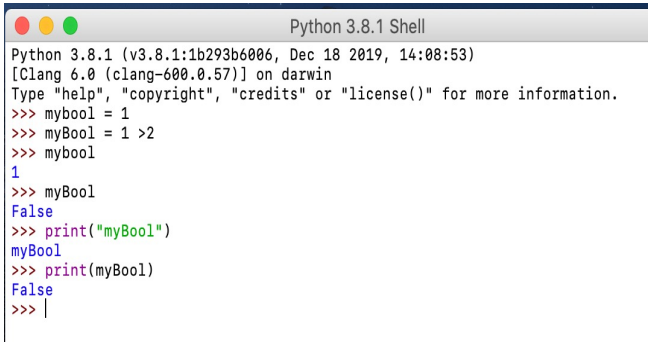
Boolean

- A computer uses yes/no, true/false, high/low, on/off logic.
- `True` or `False` are the two values stored in a Boolean, or **`bool`**, data type.
- You can directly assign either **`True`** or **`False`** (note the uppercase first letter here) to a variable, or create a statement that defines what is called a 'logical expression' that will evaluate to either True or False, e.g.

```
myBool = 1 > 2
```

← `[execute from right side first]`

The right-hand side of the expression (after the =) would evaluate to **`False`** as ***1 is not greater (>) than 2,*** and then **`False`** will be assigned to the variable `myBool`.



Finding out the type of a variable

- Sometimes you might want to know a variable's type *(what kind of data is stored in the variable)*.
- Python has a special function, **`type()`**, to tell us the data type of any value.
- E.g. (the >>> indicates the interpreter prompt, don't type it!)

```
>>> type(3)
<class 'int'>

>>> type(3.0)
<class 'float'>

>>> type("hello")
<class 'str'>

>>> myInt = 32
>>> type(myInt)
<class 'int'>
```

```
>>>
>>> type(3)
<class 'int'>
>>> type(3.0)
<class 'float'>
>>> type("hello")
<class 'str'>
>>> type(True)
<class 'bool'>
```

Declaring variables - identifiers

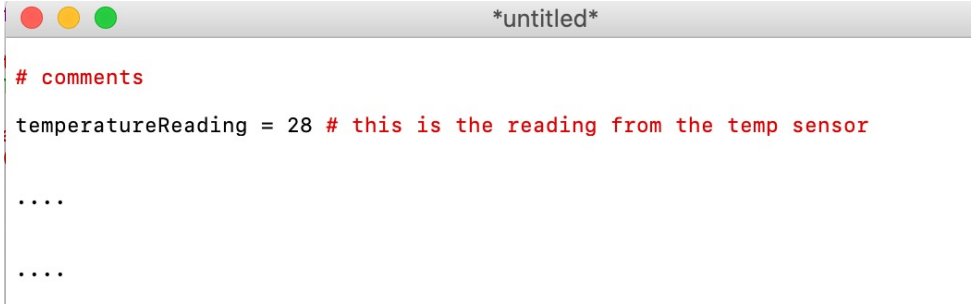
- The name of a variable is
  - Called an ***IDENTIFIER***
  - This is chosen by the programmer

```
>>> if = 5
SyntaxError: invalid syntax
>>> Jimmy Cao = 30
SyntaxError: invalid syntax
>>> 2020year = "today"
SyntaxError: invalid syntax
>>> year2020 = "today"
>>>
```

- **Rules** apply (syntax errors if broken)
  - You **cannot** use one of Python's keywords (e.g., if) as a variable name.
  - A variable name **cannot** contain spaces.
  - The **first** character **must** be one of the letters a-z, A-Z, or an underscore character (_).
  - After the first character, you *may* use the letters a-z or A-Z, the digits 0-9, or underscores.
  - Uppercase and lowercase characters are distinct. This means the variable name `ItemsOrdered` is not the same as `itemsordered`.

Declaring variables - identifiers

- Good programming principles apply – this is style
  - Normal variable identifiers **always** start with a lower-case letter
  - Identifier should be meaningful: the variable name should reflect its use
  - E.g. `temperatureReading` is better than `x3`
- *To make it easy to find variables, they should be introduced near the start of the program i.e. assign them initial values and use comments to explain their purpose*



Assignment of variables

- An **assignment statement** **changes** the value of a variable

  The syntax of the assignment statement is

  1. A variable identifier (*this is what will be changed*)
  2. Followed by **=** (this is the assignment operator and you should read it as “becomes”. It is not "equals"!).
  3. A value (which can be the result of an expression) to store in the variable

  Remember: the variable receiving a value must be on **the left side** of the **=** operator

  e.g. x = 2 *read this as "x becomes 2" or "x is assigned 2"*

- A variable can be passed as an argument to a **function**

  e.g. y = f(x) *y becomes the value of evaluating function f with the value of x as an argument*

- **Important**: You can only use a variable if a value is assigned to it

Assignment of variables

- E.g. A common error: Consider the following code:

```
# my program to demonstrate an initialisation error

temperature = temperature + 1
```

More examples,

```
>>>
>>> x = 1
>>> y = x
>>> print (y)
1
>>> print (x)
1
>>> y = z
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    y = z
NameError: name 'z' is not defined
>>> z = z + 1
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    z = z + 1
NameError: name 'z' is not defined
>>>
>>> z = 0
>>> z = z + 1
>>> print (z)
1
>>>
>>> z = 15
>>> z = z + 1
>>> print (z)
16
>>>
```

Example

- Plain language (also known as pseudo code)

  myVariable becomes 55

- Correct Syntax

```
myVariable = 55
print(myVariable)
```

What is output?

- The following will produce a different result (a sematic error – why?)

```
myVariable = 55
print("myVariable")
```

What is output?

Displaying variable values

- We can ask the python interpreter for the value of a variable by directly (manually) typing its name in the command line;

e.g. `>>> myVar` (*remember, >>> represents the prompt, don't type it*)

```
55
```

This >>>only 'works' in the console window directly, not in source code.

- Or, if writing source code, use the `print` function to display (or more formally, *evaluate*) the value for a variable

  e.g. `print(myVar)`

- You can print multiple variable values separated by commas (each value in the output will be separated by a single space)

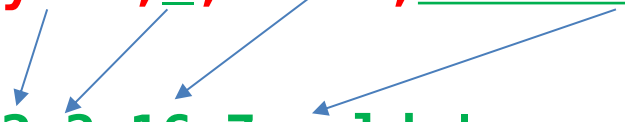
  e.g. `print(myVar,temperature,score)`

Displaying variable values

You can mix-and-match variable values and literal values

e.g.

```
myVar = 2
score = 16.7
print(myVar,2,score,"zelda's castle")
```



```
Output: 2 2 16.7 zelda's castle
```

In this case, Python doesn't care what *type of data* each item is, but it ultimately **converts** each one of them automatically to a string before it can be shown in the output.

Notice: each item is separated by one space in the output.

Displaying variable values

- We can combine multiple items together ourselves before asking `print` to output it – sometimes you want to control exactly what gets printed yourself, e.g. without extra spaces added between items etc.
- When there are two strings on either side of it, the **+** operator means to join (concatenate) the two strings together (and it is not numerical addition)

e.g. `name = "Ash" + "Ketchum"`

name contains the string "AshKetchum"

```
>>>
>>> name = "Ash" + "Ketchum"
>>> name
'AshKetchum'
>>> numberstring = str (10+6)
>>> numberstring
'16'
>>>
```

- The **`str()`** function **converts** non-string values to a string form so they can then be combined with other strings, for example convert numbers to their character representation

e.g. `numberString = str(10 + 6)` `[note: 16 convert to "16"]`

numberString contains the string "16" , which is the character '1' followed by the character '6'

Example

An example problem – we want to display the name and value of two variables:

- Plain language (*pseudo code*)

  varA becomes 5

  varB becomes 7

  Display the name and value of varA

  Display the name and value of varB

Example

```
varA = 5
varB = 7
print ("varA current value – " + str(varA))
print ("varB current value – " + str(varB))
```

Number 5 is converted to string "5"

Join two strings together

Number 7 is converted to string "7"

```
>>>
>>> varA = 5
>>> varB = 7
>>> print ("varA current value – " + varA)
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    print ("varA current value – " + varA)
TypeError: can only concatenate str (not "int") to str
>>> print ("varA current value – " + str (varA))
varA current value – 5
>>> print ("varA current value – " + str (varB))
varA current value – 7
>>>
```

Using variables

- Once you assign a value to a variable, you have to use the variable consistently
- The following example will be confusing (because it doesn't make sense to add **False** to a number – what should the result be – a number or a boolean??):

```
myVar1 = False
myVar2 = 6 + myVar1
```

```
>>>
>>> myvar1 = False
>>> myvar2 = myvar1 + 2
>>> myvar2
2
>>> myvar1 = True
>>> myvar2 = myvar1 + 2
```

False - 0
True - 1

- But - you can simply re-define the variable 'on the fly', for example the following will not cause a confusion (because it is simply adding numbers):

```
myVar1 = False
myVar1 = 11
myVar2 = 6 + myVar1
```

We've redefined (or updated) what's stored in myVar here

Assigning the same value to different variables at the same time

- Python actually allows you to assign the **same value** to several variables all at once e.g.

```
a = b = c = 10
```

This is actually evaluated starting on the far right-hand side first

i.e. *c becomes 10; b becomes c (10); a becomes b (10)*

- If the variables are all the same type, we can produce a meaningful output with the right type of expression.

```
print(a + b + c)
```

- What will the output be?

```
>>>
>>> a = b = c = 10
>>> print (a+b+c)
30
>>>
```

```
>>> c = d = e = "Jimmy"
>>> print (c+d+e)
JimmyJimmyJimmy
>>>
```

```
>>> a = "5"
>>> print (a*5)
55555
>>> b= "hello"
>>> print (b*10)
hellohellohellohellohellohellohellohellohellohello
>>>
```

Variable re-assignment

- Variables can reference ***different*** values while program is running as different assignment operations are encountered in the code
- A variable can refer to an item of any type
  - It can be assigned to one type of data and can then be re-assigned to another type of data in the future e.g. `int` then `float`
  - Other languages typically don't let you do this – usually you declare what kind of data a variable can refer to at the start of your code, and it's then never allowed to change the type e.g. *once an* `int`*, always an* `int`
  - Compilers use this information (in a process called *type checking*) to make sure you're not making mistakes in your code (for example, mistakenly assigning the wrong kind of data to a variable). Python doesn't do this.

Modifying existing variable values

- We can change an existing variable's value by simply reassigning a value to it – simply execute another assignment statement
- We need to be careful we don’t overwrite a variable that should have remained the same value
- What if we wanted to swap the values of our varA and varB variables around?

Swapping values example

wrong approach

```
>>> varA = 5
>>> varB = 7
>>> print ("the value  of varA is", varA)
the value  of varA is 5
>>> print ("the value  of varB is", varB)
the value  of varB is 7
>>> varA = varB
>>> varB = varA
>>> print ("the new value of varA is", varA)
the new value of varA is 7
>>> print ("the new value of varB is", varB)
the new value of varB is 7
>>>
```

- *Pseudo code*

1. varA becomes 5
2. varB becomes 7
3. Display the name and value of varA
4. Display the name and value of varB
5. varA becomes varB
6. varB becomes varA
7. Display the name and new value of varA
8. Display the name and new value of varB

This doesn't work – why?

- Hint – *what's the value of varA after line 5*?

Swapping values example

Correct approach

```
>>>
>>> varA = 5
>>> varB = 7
>>> print ("the value  of varA is", varA)
the value  of varA is 5
>>> print ("the value  of varB is", varB)
the value  of varB is 7
>>> temp = varA
>>> varA = varB
>>> varB = temp
>>> print ("the new value  of varA is", varA)
the new value  of varA is 7
>>> print ("the new value  of varB is", varB)
the new value  of varB is 5
>>>
```

- *Pseudo code*

1. varA becomes 5
2. varB becomes 7
3. Display the name and value of varA
4. Display the name and value of varB
5. **Create a temporary variable** called temp
6. temp becomes varA
7. varA becomes varB
8. varB becomes temp
9. Display the name and new value of varA
10. Display the name and new value of varB

line 6 - *save the value of varA before we lose it!*

Swapping values example

```
varA = 5
varB = 7

print ("varA current value – " + str(varA))
print ("varB current value – " + str(varB))
temp = varA
varA = varB
varB = temp
print ("varA new value – " + str(varA))
print ("varB new value – " + str(varB))
```

BUT ….

- Python also allows simultaneous assignment where multiple values can be assigned to **multiple variables** at the same time, meaning the temporary variable (`temp`) from our previous example isn't actually needed…
- Several values can be calculated and assigned at the same time

```
<var1>, <var2>, … = <expr1>, <expr2>, …
```

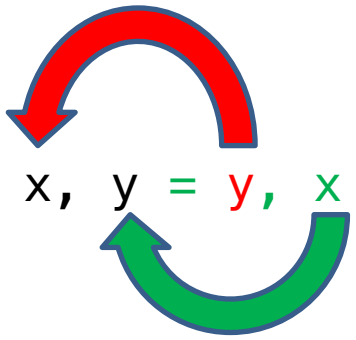
- Evaluate the expressions in the **right**-hand side (RHS) and **assign them to** the variables on the **left**-hand side (LHS) (in order)

Simultaneous assignment

- We can then swap the values of two variables quite easily in Python!

e.g. use x, y = y, x

```
>>> x = 3
>>> y = 4
>>> print (x, y)
3 4
>>> x, y = y, x
>>> print (x, y)
4 3
```




```
>>> x = 3
>>> y = 4
>>> print (x, y)
3 4
>>> x, y = y, x
>>> print (x, y)
4 3
>>>
```

Swapping values example 2

```
varA = 5
varB = 7

print ("varA current value – " + str(varA))
print ("varB current value – " + str(varB))

varA, varB = varB, varA

print ("varA new value – " + str(varA))
print ("varB new value – " + str(varB))
```

Reading input from the Keyboard

- Most programs need to read input from the user
- The built-in **`input`** function **reads input from the keyboard**
  - It **ALWAYS** returns the data read as a **string** (even if the user types in numbers!)
  - syntax: `variable = input(prompt)`
    - ***`prompt`*** is a string instructing the user to enter a value e.g.
      `name = input("Please enter your name:")`
  - `input` does not automatically display a space after the prompt, so you usually must make it obvious to the user you're waiting for some data – typically you might use a colon `:` character and a space

This is clearer that you're typing here with prompt..

```
Please enter your name: Fred
ur nameFred
```

No separation of the prompt from the user's input

```
>>> name = input ("please enter your name:")
please enter your name:Jimmy
>>> name
'Jimmy'
>>> temp = input ("please enter the temperature today:")
please enter the temperature today:25
>>> temp
'25'
>>> type(temp)
<class 'str'>
>>>
```

Reading **numbers** with the `input` Function

- The purpose of an `input` statement is usually to get input from the user and **store it into a variable** to use later, otherwise if you don't immediately use the data it's *lost*
- **Remember** – the `input` function always returns a string
- Python has built-in functions to convert between data types
  - `int(item)` converts `item` to an `int`
  - `float(item)` converts `item` to a `float`
  - This type conversion only works if `item` is a valid numeric value, otherwise an error occurs e.g.

    `int("3")` returns `3`

    `int("banana")` is an error (*how work out what an integer relate*

```
>>> temp = input ("please enter the temperature today:")
please enter the temperature today:25
>>> temp
'25'
>>> type(temp)
<class 'str'>
>>>
>>> int(temp)
25
>>> temp = input ("please enter the temperature today:")
please enter the temperature today:25.8
>>> temp
'25.8'
>>> int(temp)
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    int(temp)
ValueError: invalid literal for int() with base 10: '25.8'
>>> float(temp)
25.8
>>>
```

Reading numbers with the `input` Function

***variable* = int(input(*prompt*))**

We can immediately try to convert the string value given to us by input to an integer using the `int` function

1. First the *prompt* value is printed
2. The input part causes the program to stop and wait for the user to enter a value and press the ***enter*** key
3. The expression that was entered by the user is then evaluated to try to turn it from a string of characters into an **`int`** value (*if it can't, the program stops with an error!*)
4. The value is assigned to the *variable*.

input examples

```
person = input('Enter your name: ')
print('Hello',person,'!')
```

If we enter John, the result would be:

```
Hello John !
```

```
>>> person = input ("enter your name:")
enter your name:John
>>> print ("Hello" , person , "!")
Hello John !
>>> print ("Hello" + person + "!")
HelloJohn!
>>> print ("Hello " + person + "!")
Hello John!
>>> print('Hello ', person, '!', sep='')
Hello John!
>>>
```

Hmm.. there should not be a space before the '!'. Two ways to fix this:

1. The + operation on strings

```
print('Hello ' + person + '!')
```

```
>>> print('Hello ', person, '!', sep='')
Hello John!
>>> print('Hello ', person, '!', sep=',')
Hello ,John,!
>>> print('Hello ', person, '!', sep=' here')
Hello  hereJohn here!
>>>
```

2. Use a special argument (**sep**) to the **print** function:

```
print('Hello ',person,'!',sep='')
```

'' *is an empty string (no chars between the quotes). Here it means* ***'don't add any separators between items'****. If you use this option, you have to add spaces manually yourself when you need a space in the output*

input examples

```
xString = input("Enter a number: ")
x = int(xString)
```

Two steps

OR

```
x = int(input("Enter a number: "))
```

One step

```
x = int(input("Enter an integer: "))
y = int(input("Enter another integer: "))
sum = x + y
print('The sum of ',x,' and ',y,' is ',sum,'.',sep='')
```

Is this numerical addition or string concatenation?

```
>>> x = int(input("Enter an integer: "))
Enter an integer: 5
>>> y = int(input("Enter another integer: "))
Enter another integer: 10
>>> sum = x + y
>>> sum
15
>>> type(sum)
<class 'int'>
>>> print('The sum of ',x,' and ',y,' is ',sum,'.',sep='')
The sum of 5 and 10 is 15.
>>>
```

Displaying multiple items with the `print` function

Now you know

- Python allows one to display multiple items with a single call to `print`:

  ```
  print(<expr>, <expr>, …, <expr>)
  ```

  - Items are separated by **commas** when passed as arguments
  - Arguments/variables are displayed in the order they are passed to the function
  - Items are automatically separated by a **space** when displayed on the output (you can use **+** or **`sep=''`** to aviod a space)
  - **`print()`** (without any arguments/variables) will print a single blank line.

Questions?