COMP10001 UNABRIDGED NOTES

Worksheet 0 – Blockly

- Programs are just a set of **statements** which are completed in a **sequence**
- ie. Turn -> go forward __ steps -> …

Looping

- An alternative to repetitive coding- it will repeat a sequence of steps a set amount of times. This is a form of **control flow**
- The steps which are being repeated are said to be *inside* the loop
- The number of **iterations** is the number of times that the loop is completed
- We are currently programming **for** loops, a specific type of loop
- **Nesting** Statements: Putting compound statements inside other compound statements (ie. Putting a loop inside a loop)

Conditioning/Control

- Allowing a continuing of steps **if** a set of terms is true. The next statement in a sequence may be depended on a condition (ie. End if completed 3 times already)
- Can be structured like **if** and **else**, whereby if the 'if' steps aren't completed then the 'else' steps will be.

Functions (abstraction)

- Defining a set of statements/steps, giving it a different input, so, when later **called** upon by calling the defined name, the set of statements will be completed
- Defining alone will not complete the step sequence
- You can include **variables/arguments** in the defined function, meaning that when you call upon the function later, you will have to specify the desired variable value (ie. 'side length')

Recursion

- blocks of code that call themselves with different inputs
- A step within a sequence can call upon the sequence again itself, sequence-ception

Worksheet 1 – Calculations & Variables

Print

- One of python's **built-in** functions
- Use **delimiters**, in this case, quotation marks, to show the start and end of the **string** of text. This will distinguish it from any other type of python code.

- If the string of text contains a " or a ' it is best to delimit the string with the other type of quotation mark.
- You can 'print' multiple words stuck together using * or + outside of the delimiters. When applied to strings, these operations are referred to as **concatenation**.
- We can print two things at once by print(a**,** b)
- 3 x " at the end of a string will end the string, at the start of the string, all after """ will be included in the string. You *can*, therefore, use " in a string if you also use " around the outside
- To include quotation marks or slashes in a string, use a \ before them. Ie. \", \', \\, \t (tab), \n (newline)
- To work out the length of the string, use **print(len("__"))**. (To work out the length, or number of digits of, an integer, convert to str first- print(len(str(__))).)

Variables

- You can set a string or a value by defining it, or setting a variable equal to it. Put the variable/definition to the left of a **=** sign. Ie. Message = Hello World
- = in this case is not "is equal to"; it can be translated to "assign to".
- The LHS can only be one thing (ie. Not a + 1)
- We can **stack** variables by using multiple = signs on the one line. Again, only the very RHS can be only one thing.
- You can **assign** a different value to a variable later on in a code, you can even use the initial variable to change itself (ie. Msg = msg * 2)
- If a=0, and you assign b=a, then change a to =1, b will still =0. Python is an **imperative** language, meaning the values of variables can only be changed through **re**assignment. For c = d + a, look for the last time we had d and a on the LHS, and use that assignment. This is different for item **mutation** (as opposed to **reassignment**), in which case b would mutate in the same way a has.
- To avoid this, we could make an immutable, and when assigning b to it, convert it to a mutable type, so b is mutable but a still is not
- **Literals**: represent only what they literally means. Includes numbers, strings.
- NAMING VARIABLES RULES:
  1) Must start with a letter or underscore (casing is significant)
  2) Can have only a letter, underscore or number in its name
  3) `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `False`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `None`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `True`, `try`, `while`, `with`, `yield` cannot be used, as they are built-in prompts (**keywords**) for python. 'print' isn't a keyword, but using it will lose the original function of the word.

Calculate

- put **print** around the calculation again
- BIDMAS occurs, and calculations which occur first as a result of this hierarchy are said to hold **higher precedence**
- **Parentheses** refer to brackets

Operator	Description
*=	`c *= 2` is the same as `c = c * 2`
/=	`c /= 2` is the same as `c = c / 2`
+=	`c += 2` is the same as `c = c + 2`
-=	`c -= 2` is the same as `c = c - 2`
//=	`c //= 2` is the same as `c = c // 2`
%=	`c %= 2` is the same as `c = c % 2`
**=	`c **= 2` is the same as `c = c**2`

- ** means to the power of
- 1e+10 means 10 to the power of 10, multiplied by 1. This will be considered **float**
- Division of two integers, like 3 / 4, will give a float answer
- As such, // means divide, and round down answer to integer
- % is modulo (finds remainder after a division of one number by another). It is on the same level as * and / in the BIDMAS hierarchy
- Python 2 approaches this differently
- Special character _ refers to the answer value of the previous calculation.
- For float calculations, some numbers will inevitable have **rounding error**, whereby the value is marginally inaccurate.

Comment

- Putting a **#** before a length of text means that the text is a comment, and is not read by python, so is intended for human reading only

syntax highlighting (i.e. using different colours to represent different constructs in your code) can be a strong visual cue when browsing over code, in distinguishing between code and comments.

```
# This is a comment. Below is a line of code:
print("This is a line of code, not a comment")
# The next line looks like a command but isn't!
#print("This won't print if you try to run it!")
```

If you want a command to continue over the line, insert a backslash (****) called an **escape character** at the end of the line.

Input

- The input function requires the user interface to input text after a statement, the input thereafter designed and set up to be included in a later line of code. (ie. Name = input("What is you name: ") // print ("hi", name)
- The comma here allows distinction to the separate parts of the print
- Inputs will only deal with strings, so calculations between inputs will only be concatenations
- We can avoid this dilemma by putting float or int around the input function. ie print(int(input('what is your age? ')))

- *Worksheet 2, Type conversion exercise 2*

Worksheet 2 – Types

Operands: the numbers, values, variables which we operate on.

We can use expressions (ie. Y, where y = 3) in values where we would otherwise need values, however, we cannot include assignment statements (ie. Y = 3) in these places.

Types

- Every object has a 'type' which defines what functions can be applied to it, as well as defining its semantics
- The types we will see are: **complex** (complex numbers)

Type	Description
`int`	For whole numbers eg: `-3`, `-5`, or `10`
`float`	For real numbers eg: `-3.0`, `0.5`, or `3.14159`
`bool`	The Boolean type. For storing `True` and `False` (only those two values; Booleans allow for no grey areas!).
`str` (= "string")	For chunks of text, e.g.: `"Hello, I study Python"`
`tuple`	For combinations of objects, e.g.: `(1, 2, 3)` or `(1.0, "hello", "frank")`
`list`	A more powerful way of storing lists of objects, e.g. `[1, 3, 4]` or `[1.0, "hello", "frank"]`
`dict`	We will see this later ... maybe you can guess what it does, e.g. `{"bob": 34, "frankenstein": 203}`

- **To determine the object's type:**

```
>>> print(type(1))
<type 'int'>
>>> print(type(1.0))
<type 'float'>
```

- **Type Casting:** we can convert a literal/variable to a different type to what it is

```
>>> print(float(1))
1.0
>>> print(int(1.5))
1
>>> int('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
```