MODULE-1
THE WAY OF THE PROGRAM, VARIABLES, EXPRESSIONS, STATEMENTS, ITERATION, AND FUNCTIONS

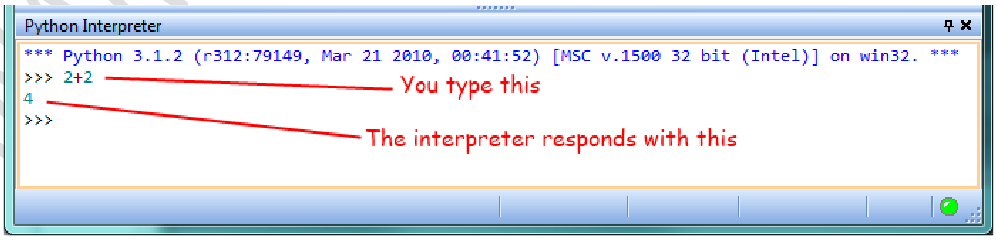
SYLLABUS:
The way of the program: *The Python programming language, what is a program? What is debugging? Syntax errors, Runtime errors, Semantic errors, Experimental debugging.*
Variables, Expressions and Statements: *Values and data types, Variables, Variable names and keywords, Statements, Evaluating expressions, Operators and operands, Type converter functions, Order of operations, Operations on strings, Input, Composition, The modulus operator.*
Iteration: *Assignment, updating variables, the for loop, the while statement, The Collatz 3n + 1 sequence, tables, two-dimensional tables, break statement, continue statement, paired data, Nested Loops for Nested Data.*
Functions: *Functions with arguments and return values.*

CHAPTERS: *1.1-1.7, 2.1-2.12, 3.3, 4.4, 4.5*

TEXT BOOK: *Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers- How to think like a computer scientist: learning with python 3. Green Tea Press, Wellesley, Massachusetts, 2020*

THE PYTHON PROGRAMMING LANGUAGE

- Python is an example of a high-level language; other high-level languages are C++, PHP, Pascal, C#, and Java.
- Low-level languages, also known as machine languages or assembly languages, are specialized programming languages that computers can only execute.
- High-level languages require translation into a more suitable format before they can run, as computers can only execute programs written in low-level languages.
- High-level languages are commonly used in programming due to their advantages, such as being easier to write, shorter, easier to read, and more likely to be correct. Additionally, these languages are portable, allowing them to run on various computers with minimal modifications.
- The engine that translates and runs Python is called the Python Interpreter: There are two ways to use it: immediate mode and script mode. In immediate mode, you type Python expressions into the Python Interpreter window, and the interpreter immediately shows the result:



```
*** Python 3.1.2 (r312:79149, Mar 21 2010, 00:41:52) [MSC v.1500 32 bit (Intel)] on win32. ***
>>> 2+2
4
>>>
```

- The >>> is called the Python prompt. The interpreter uses the prompt to indicate that it is ready for instructions. For example, typed 2 + 2, and the interpreter evaluated our expression, and replied 4, and on the next line it gave a new prompt, indicating that it is ready for more input.
- Alternatively, you can write a program in a file and use the interpreter to execute the contents of the file. Such a file is called a script. Scripts have the advantage that they can be saved to disk, printed, and so on.

- Working directly in the interpreter is ideal for testing short code as it provides immediate feedback, similar to using scratch paper for problem-solving. For longer code, use a script.
- To write a script, a text editor is essential, a program on your computer that changes text files, not a word processor like MicrosoftWord or LibreOffice Writer. Examples include Notepad, Notepad++, vim, emacs, and sublime.
- Python and other programming languages have development environments, which include a text editor and interpreter interaction. Examples include Spyder, Thonny, and IDLE. Some development environments run in the browser, like Jupyter Notebook. These environments, sometimes called Integrated Development Environments or IDEs, are essential for efficient programming.
- The choice of development environment is quite personal. The important thing to remember is that Python itself does not care in what editor is used to write your code. As long as you write correct syntax (with the right tabs and spaces) Python can run your program. The editor is only there to help you.

WHAT IS A PROGRAM?

- A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or (strangely enough) compiling a program.
- The details of a task may vary across different languages, but basic instructions are generally present in almost all languages:
  - **Input:** Get data from the keyboard, a file, or some other device such as a sensor.
  - **Output:** Display data on the screen or send data to a file or other device such as a motor.
  - **Math:** Perform basic mathematical operations like addition and multiplication.
  - **Conditional Execution:** Check for certain conditions and execute the appropriate sequence of statements.
  - **Repetition:** Perform some action repeatedly, usually with some variation.
- Programming is the process of breaking down a complex task into smaller subtasks until they are simple enough to be performed using sequences of basic instructions. Every program, regardless of complexity, is composed of instructions similar to these.

WHAT IS DEBUGGING?

- Programming is a complex process, and because it is done by human beings, it often leads to errors. Programming errors are called bugs and the process of tracking them down and correcting them is called debugging.
- Use of the term bug to describe small engineering difficulties dates back to at least 1889, when Thomas Edison had a bug with his phonograph.
- Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

SYNTAX ERRORS

- Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message.
- Syntax refers to the structure of a program and the rules about that structure.

- In English, a sentence must start with a capital letter and end with a period. This sentence contains a syntax error. Most readers find these errors minor, allowing them to read E. E. Cummings' poetry without issues.
- Python is not so forgiving because if there is a single syntax error anywhere in your program, Python will display an error message and quit, and will not be able to run the program.
- Initially, in a programming career may spend a significant amount of time addressing syntax errors, but with experience, later make fewer mistakes and find them faster.

RUNTIME ERRORS

- The second type of error is a runtime error, so called because the error does not appear until you run the program.
- These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened. Runtime errors are rare in the simple programs.

SEMANTIC ERRORS

- A semantic error is a third type of error where a program will run successfully without generating error messages, but it will not perform the correct task, instead performing what you have instructed it to do.
- The program you wrote is not the desired one due to incorrect semantics. Identifying semantic errors is challenging as it requires examining the program's output and determining its purpose, making it difficult to work backwards.

EXPERIMENTAL DEBUGGING

- One of the most important skills is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.
- Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, When you have eliminated the impossible, whatever remains, however improbable, must be the truth. (A. Conan Doyle, The Sign of Four)
- For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does something and make small modifications, debugging them as you go, so that you always have a working program.
- For example, Linux is an operating system kernel that contains millions of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, one of Linus's earlier projects was a program that would switch between displaying AAAA and BBBB. This later evolved to Linux (The Linux Users' Guide Beta Version 1).

VALUES AND DATA TYPES

- A value is one of the fundamental things like a letter or a number, that a program manipulates. Thes values are classified into different classes, or data types: For example, 4 is an integer, and "Hello, World!" is a string, so-called because it contains a string of letters. Hence identifying strings is easy because they are enclosed in quotation marks.

- If not sure what class a value falls into, Python has a function called type which can tell you.

```
>>> type("Hello, World!")
<class 'str'>
>>> type(17)
<class 'int'>
```

- Strings belong to the class str, integers belong to the class int and numbers with a decimal point belong to a class called float, because these numbers are represented in a format called floating-point. Here at this stage, treat the words class and type interchangeably.

```
>>> type(3.2)
<class 'float'>
```

- What about values like "17" and "3.2"? They look like numbers, but they are in quotation marks like strings. But, They're strings!

```
>>> type("17")
<class 'str'>
>>> type("3.2")
<class 'str'>
```

- Strings in Python can be enclosed in either single quotes (') or double quotes ("), or three of each (' ' ' or " " ")

```
>>> type('This is a string.')
<class 'str'>
>>> type("And so is this.")
<class 'str'>
>>> type("""and this.""")
<class 'str'>
>>> type('''and even this...''')
<class 'str'>
```

- Double quoted strings can contain single quotes inside them, as in "Bruce's beard", and single quoted strings can have double quotes inside them, as in 'The knights who say "Ni!"'. Strings enclosed with three occurrences of either quote symbol are called triple quoted strings. They can contain either single or double quotes:

```
>>> print("""Oh no", she exclaimed, "Ben's bike is broken!""")
"Oh no", she exclaimed, "Ben's bike is broken!"
>>>
```

- Triple quoted strings can even span multiple lines:

```
>>> message = """This message will
... span several
... lines."""
>>> print(message)
This message will
```

```
span several
lines.
>>>
```

- Python doesn't care whether you use single or double quotes or the three-of-a-kind quotes to surround your strings: once it has parsed the text of your program or command, the way it stores the value is identical in all cases, and the surrounding quotes are not part of the value. But when the interpreter wants to display a string, it has to decide which quotes to use to make it look like a string. Hence, Python language designers usually chose to surround their strings by single quotes.

```
>>> 'This is a string.'
'This is a string.'
>>> """And so is this."""
'And so is this.'
```

- When you type a large integer, you might be tempted to use commas between groups of three digits, as in 42,000. The same goes for entering Dutch-style floating point numbers using a comma instead of a decimal dot. This is not a legal integer in Python, but it does mean something else, which is legal:

```
>>> 42000
42000
>>> 42,000
(42, 0)
```

- Because of the comma, Python chose to treat this as a pair of values. But, remember not to put commas or spaces in integers, no matter how big they are.

VARIABLES

- One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value. The assignment statement gives a value to a variable:

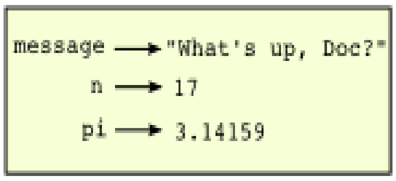
```
>>> message = "What's up, Doc?"
>>> n = 17
>>> pi = 3.14159
```

- This example makes three assignments. The first assigns the string value "What's up, Doc?" to a variable named message. The second gives the integer 17 to n, and the third assigns the floating-point number 3.14159 to a variable called pi.
- The assignment token, =, should not be confused with equals, which uses the token ==. The assignment statement binds a name, on the left-hand side of the operator, to a value, on the right-hand side. Basically, an assignment is an order, and the equals operator can be read as a question mark. This is why an error if when entered:

```
>>> 17 = n
File "<interactive input>", line 1
SyntaxError: can't assign to literal
```

- A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a state snapshot because it shows what state each of the variables is in at a particular instant in time. (Think of it as the variable's state of mind).

- Some editors and programming environments do this for you, and allow you to click through the state of the program saving you some paper. This diagram shows the result of executing the assignment statements:



- If asked the interpreter to evaluate a variable, it will produce the value that is currently linked to the variable:

```
>>> message
'What's up, Doc?'
>>> n
17
>>> pi
3.14159
```

- We use variables in a program to "remember" things, perhaps the current score at the football game. But variables are variable. This means they can change over time, just like the scoreboard at a football game. You can assign a value to a variable, and later assign a different value to the same variable. (This is different from maths. In maths, if you give 'x' the value 3, it cannot change to link to a different value half-way through your calculations!)

```
>>> day = "Thursday"
>>> day
'Thursday'
>>> day = "Friday"
>>> day
'Friday'
>>> day = 21
>>> day
21
```

- Changed the value of day three times, and on the third assignment we even made it refer to a value that was of a different type.
- A great deal of programming is about having the computer remember things, e.g. The number of missed calls on your phone, and then arranging to update or change the variable when you miss another call.

VARIABLE NAMES AND KEYWORDS

- Variable names can be arbitrarily long. They can contain both letters and digits, but they have to begin with a letter or an underscore. Although it is legal to use uppercase letters, by convention we don't. Remember that case matters. Bruce and bruce are different variables.
- The underscore character (_) can appear in a name. It is often used in names with multiple words, such as my_name or price_of_tea_in_china.
- There are some situations in which names beginning with an underscore have special meaning, so a safe rule for beginners is to start all names with a letter. If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = "big parade"
SyntaxError: invalid syntax
>>> more$ = 1000000
SyntaxError: invalid syntax
>>> class = "Computer Science 101"
SyntaxError: invalid syntax
```

- 76trombones is illegal because it does not begin with a letter. more$ is illegal because it contains an illegal character, the dollar sign. But what's wrong with class? It turns out that class is one of the Python keywords. Keywords define the language's syntax rules and structure, and they cannot be used as variable names.
- Python has thirty-something keywords (and every now and again improvements to Python introduce or eliminate one or two):

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

- Programmers generally choose names for their variables that are meaningful to the human readers of the program —they help the programmer document, or remember, what the variable is used for.
- Beginners sometimes confuse "meaningful to the human readers" with "meaningful to the computer". So they'll wrongly think that because they've called some variable average or pi, it will somehow magically calculate an average, or magically know that the variable pi should have a value like 3.14159. No! The computer doesn't understand what intend the variable to mean.
- So you'll find some instructors who deliberately don't choose meaningful names when they teach beginners not because we don't think it is a good habit, but because we're trying to reinforce the message that the programmer, must write the program code to calculate the average, and must write an assignment statement to give the variable pi the value want it to have.

```
e = 3.1415
ray = 10
size = e * ray ** 2

pi = 3.1415
radius = 10
area = pi * radius ** 2
```

- The above two snippets do exactly the same thing, but the bottom one uses the right kind of variable names. For the computer there is no difference at all, but for a human, using the names and letters that are part of the conventional way of writing things make all the difference in the world. Using e instead of pi completely confuses people, while computers will just perform the calculation!

STATEMENTS

- A statement is an instruction that the Python interpreter can execute. We have only seen the assignment statement so far.

- Some other kinds of statements that we'll see shortly are while statements, for statements, if statements, and import statements. (There are other kinds too!)
- When typed a statement on the command line, Python executes it. Statements don't produce any result.

EVALUATING EXPRESSIONS

- An expression is a combination of values, variables, operators, and calls to functions. If typed an expression at the Python prompt, the interpreter evaluates it and displays the result:

```
>>> 1 + 1
2
>>> len("hello")
5
```

- In this example len is a built-in Python function that returns the number of characters in a string.
- The evaluation of an expression produces a value, which is why expressions can appear on the right-hand side of assignment statements. A value all by itself is a simple expression, and so is a variable.

```
>>> 17
17
>>> y = 3.14
>>> x = len("hello")
>>> x
5
>>> y
3.14
```

OPERATORS AND OPERANDS

- Operators are special tokens that represent computations like addition, multiplication and division. The values the operator uses are called operands.
- The following are all legal Python expressions whose meaning is more or less clear:

```
20+32        hour-1       hour*60+minute       minute/60       5**2      (5+9)*(15-7)
```

- The tokens +, -, and *, and the use of parenthesis for grouping, mean in Python what they mean in mathematics. The asterisk (*) is the token for multiplication, and ** is the token for exponentiation.

```
>>> 2 ** 3
8
>>> 3 ** 2
9
```

- When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed. Addition, subtraction, multiplication, and exponentiation all do what expected.
- Example: so let us convert 645 minutes into hours:

```
>>> minutes = 645
>>> hours = minutes / 60
>>> hours
```

```
10.75
```

- In Python 3, the division operator / always yields a floating point result. What might have wanted to know was how many whole hours there are, and how many minutes remain. Python gives us two different flavors of the division operator. The second, called floor division uses the token //. Its result is always a whole number and if it has to adjust the number it always moves it to the left on the number line. So 6 // 4 yields 1, but -6 // 4 yields -2.

```
>>> 7 / 4
1.75
>>> 7 // 4
1
>>> minutes = 645
>>> hours = minutes // 60
>>> hours
10
```

- Choose the correct flavor of the division operator. If working with expressions where need floating point values, use the division operator that does the division accurately.

TYPE CONVERTER FUNCTIONS

- Here, three more Python functions, int, float and str, which will (attempt to) convert their arguments into types int, float and str respectively, call these type converter functions.
- The int function can take a floating-point number or a string, and turn it into an int. For floating point numbers, it discards the decimal portion of the number — a process we call truncation towards zero on the number line. Let's see this in action:

```
>>> int(3.14)
3
>>> int(3.9999)     # This doesn't round to the closest int!
3
>>> int(3.0)
3
>>> int(-3.999)     # Note that the result is closer to zero
-3
>>> int(minutes / 60)
10
>>> int("2345")     # Parse a string to produce an int
2345
>>> int(17)         # It even works if arg is already an int
17
>>> int("23 bottles")
```

- This last case doesn't look like a number, gives:

```
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23 bottles'
```

- The type converter float can turn an integer, a float, or a syntactically legal string into a float:

```
>>> float(17)
17.0
>>> float("123.45")
123.45
```

- The type converter str turns its argument into a string:

```
>>> str(17)
'17'
>>> str(123.45)
'123.45'
```

ORDER OF OPERATIONS

- When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym PEMDAS is a useful way to remember the order of operations:
  1. Parentheses have the highest precedence and can be used to force an expression to evaluate in the order wanted. Since expressions in parentheses are evaluated first, 2 * (3-1) is 4, and (1+1)**(5-2) is 8. It can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even though it doesn't change the result.
  2. Exponentiation has the next highest precedence, so 2**1+1 is 3 and not 4, and 3*1**3 is 3 and not 27.
  3. Multiplication and both Division operators have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence. So 2*3-1 yields 5 rather than 4, and 5-2*2 is 1, not 6.
  4. Operators with the same precedence are evaluated from left-to-right. In algebra, they are left-associative. So in the expression 6-3+2, the subtraction happens first, yielding 3. Then add 2 to get the result 5. If the operations had been evaluated from right to left, the result would have been 6-(3+2), which is 1. (*The acronym PEDMAS could mislead you to thinking that division has higher precedence than multiplication, and addition is done ahead of subtraction, don't be misled. Subtraction and addition are at the same precedence, and the left-to-right rule applies.*)
- Due to some historical quirk, an exception to the left-to-right left-associative rule is the exponentiation operator **, so a useful hint is to always use parentheses to force exactly the order wanted when exponentiation is involved:

```
>>> 2 ** 3 ** 2 # The right-most ** operator gets done first!
512
>>> (2 ** 3) ** 2 # Use parentheses to force the order you want!
64
```

- The immediate mode command prompt of Python is great for exploring and experimenting with expressions like this.

OPERATIONS ON STRINGS

- In general, it cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that message has type string):

```
>>> message - 1 # Error
>>> "Hello" / 123 # Error
>>> message * "Hello" # Error
>>> "15" + 2 # Error
```

- Interestingly, the + operator does work with strings, but for strings, the + operator represents concatenation, not addition. Concatenation means joining the two operands by linking them end-to-end. For example:

```
fruit = "banana"
baked_good = " nut bread"
print(fruit + baked_good)
```

- The output of this program is banana nut bread. The space before the word nut is part of the string, and is necessary to produce the space between the concatenated strings.
- The * operator also works on strings; it performs repetition. For example, 'Fun'*3 is 'FunFunFun'. One of the operands has to be a string; the other has to be an integer.
- On one hand, this interpretation of + and * makes sense by analogy with addition and multiplication. Just as 4*3 is equivalent to 4+4+4, we expect "Fun"*3 to be the same as "Fun"+"Fun"+"Fun", and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition and multiplication have that string concatenation and repetition do not?

INPUT

- There is a built-in function in Python for getting input from the user:

```
name = input("Please enter your name: ")
```

- The user of the program can enter the name and click OK, and when this happens the text that has been entered is returned from the input function, and in this case assigned to the variable name.
- Even if asked the user to enter their age, you would get back a string like "17". It would be a job, as the programmer, to convert that string into a int or a float, using the int or float converter functions we saw earlier.

COMPOSITION

- We have looked at the elements of a program: variables, expressions, statements, and function calls, in isolation, without talking about how to combine them.
- One of the most useful features of programming languages is their ability to take small building blocks and compose them into larger chunks.
- For example, we know how to get the user to enter some input, we know how to convert the string we get into a float, we know how to write a complex expression, and we know how to print values. Let's put these together in a small four-step program that asks the user to input a value for the radius of a circle, and then computes the area of the circle from the formula:

$$\textbf{Area} = \pi R^2$$

- Firstly, we'll do the four steps one at a time:

```
response = input("What is your radius? ")
r = float(response)
area = 3.14159 * r**2
```

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

- Now let's compose the first two lines into a single line of code, and compose the second two lines into another line of code.

```
r = float( input("What is your radius? ") )
print("The area is ", 3.14159 * r**2)
```

- If really wanted to be tricky, we could write it all in one statement:

```
print("The area is ", 3.14159*float(input("What is your radius?"))**2)
```

- Such compact code may not be most understandable for humans, but it does illustrate how we can compose bigger chunks from our building blocks. If ever in doubt about whether to compose code or fragment it into smaller steps, try to make it as simple as you can for the human to follow. Choice would be the first case above, with four separate steps.

THE MODULUS OPERATOR

- The modulus operator works on integers (and integer expressions) and gives the remainder when the first number is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators. It has the same precedence as the multiplication operator.

```
>>> q = 7 // 3                    # This is integer division operator
>>> print(q)
2
>>> r = 7 % 3
>>> print(r)
1
```

So 7 divided by 3 is 2 with a remainder of 1.

- The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another—if x % y is zero, then x is divisible by y.
- Also, can extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly x % 100 yields the last two digits.
- It is also extremely useful for doing conversions, say from seconds, to hours, minutes and seconds. So let's write a program to ask the user to enter some seconds, and we'll convert them into hours, minutes, and remaining seconds.

```
total_secs = int(input("How many seconds, in total?"))
hours = total_secs // 3600
secs_still_remaining = total_secs % 3600
minutes = secs_still_remaining // 60
secs_finally_remaining = secs_still_remaining % 60
print("Hrs=", hours, " mins=", minutes,
                    "secs=", secs_finally_remaining)
```

ITERATION

- Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

- Repeated execution of a set of statements is called iteration. Because iteration is so common, Python provides several language features to make it easier, already seen the for statement. This is the form of iteration likely be using most often.
- But here we're going to look at the while statement — another way to have your program do iteration, useful in slightly different circumstances.

ASSIGNMENT

- It is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value), because the first time airtime_remaining is printed, its value is 15, and the second time, its value is 7.

```
airtime_remaining = 15
print(airtime_remaining)
airtime_remaining = 7
print(airtime_remaining)
```

The output of this program is:

```
15
7
```

- It is especially important to distinguish between an assignment statement and a Boolean expression that tests for equality. Because Python uses the equal token (=) for assignment, it is tempting to interpret a statement like a = b as a Boolean test. Unlike mathematics, it is not! Remember that the Python token for the equality operator is ==.
- Note too that an equality test is symmetric, but assignment is not. For example, if a == 7 then 7 == a. But in Python, the statement a = 7 is legal and 7 = a is not.
- In Python, an assignment statement can make two variables equal, but because further assignments can change either of them, they don't have to stay that way:

```
a = 5
b = a                    # After executing this line, a and b are now equal
a = 3                    # After executing this line, a and b are no longer equal
```

- The third line changes the value of a but does not change the value of b, so they are no longer equal. (In some programming languages, a different symbol is used for assignment, such as <- or :=, to avoid confusion.
- Some people also think that variable was an unfortunae word to choose, and instead we should have called them assignables. Python chooses to follow common terminology and token usage, also found in languages like C, C++, Java, and C#, so we use the tokens = for assignment, == for equality, and we talk of variables.

UPDATING VARIABLES

- When an assignment statement is executed, the right-hand side expression (i.e. the expression that comes after the assignment token) is evaluated first. This produces a value. Then the assignment is made, so that the variable on the left-hand side now refers to the new value.
- One of the most common forms of assignment is an update, where the new value of the variable depends on its old value. Deduct 40 cents from my airtime balance, or add one run to the scoreboard.

```
n = 5
n = 3 * n + 1
```

- Line 2 means get the current value of n, multiply it by three and add one, and assign the answer to n, thus making n refer to the value. So after executing the two lines above, n will point/refer to the integer 16.
- If you try to get the value of a variable that has never been assigned to, you'll get an error:

```
>>> w = x + 1
Traceback (most recent call last):
File "<interactive input>", line 1, in
NameError: name 'x' is not defined
```

- Before updating a variable, have to initialize it to some starting value, usually with a simple assignment:

```
runs_scored = 0
...
runs_scored = runs_scored + 1
```

- Line 3, updating a variable by adding 1 to it, is very common. It is called an increment of the variable; subtracting 1 is called a decrement. Sometimes programmers also talk about bumping a variable, which means the same as incrementing it by 1. This is commonly done with the += operator.

```
runs_scored = 0
...
runs_scored += 1
```

THE FOR LOOP REVISITED

- In a for loop processes each item in a list, each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed:

```
for friend in ["Joe", "Zoe", "Zuki", "Thandi", "Paris"]:
invite = "Hi " + friend + ". Please come to my party!"
print(invite)
```

- Running through all the items in a list is called traversing the list, or traversal.
- Write some code now to sum up all the elements in a list of numbers. Do this by hand first, and try to isolate exactly what steps you take. Later find you need to keep some "running total" of the sum so far, either on a piece of paper, in your head, or in your calculator.
- Remembering things from one step to the next is precisely why we have variables in a program: so we'll need some variable to remember the "running total". It should be initialized with a value of zero, and then we need to traverse the items in the list. For each item, we'll want to update the running total by adding the next number to it.

```
numbers = [5, 6, 32, 21, 9]
running_total = 0
for number in numbers:
running_total = running_total + number
print(running_total)
```

THE WHILE STATEMENT

➢ Here is a fragment of code that demonstrates the use of the while statement:

```
while <CONDITION>:
<STATEMENT>
n = 6
current_sum = 0
i = 0
while i <= n:
current_sum += i
i += 1
print(current_sum)
```

➢ It means, while i is less than or equal to n, continue executing the body of the loop. Within the body, each time, increment i. When i passes n, return your accumulated sum. In other words: while <CONDITION> is True, <STATEMENT> is executed. Of course, this example could be written more concisely as sum(range(n + 1)) because the function sum already exists.

➢ More formally, here is precise flow of execution for a while statement:
- Evaluate the condition at line 5, yielding a value which is either False or True.
- If the value is False, exit the while statement and continue execution at the next statement (line 8 in this case).
- If the value is True, execute each of the statements in the body (lines 6 and 7) and then go back to the while statement at line 5.

➢ The body consists of all of the statements indented below the while keyword. Notice that if the loop condition is False the first time we get loop, the statements in the body of the loop are never executed.

➢ The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an infinite loop.

➢ In the case here, we can prove that the loop terminates because we know that the value of n is finite, and we can see that the value of i increments each time through the loop, so eventually it will have to exceed n. In other cases it is not so easy, maybe even impossible, to tell if the loop will ever terminate.

➢ What we will notice here is that the while loop is more work for the programmer — than the equivalent for loop. When using a while loop one has to manage the loop variable yourself: give it an initial value, test for completion, and then make sure you change something in the body so that the loop terminates. By comparison, here is an equivalent snippet that uses for instead:

```
n = 6
current_sum = 0
for i in range(n+1):
current_sum += i
print(current_sum)
```

➢ Notice the slightly tricky call to the range function — we had to add one onto n, because range generates its list up to but excluding the value you give it. It would be easy to make a programming mistake and overlook this.

THE COLLATZ 3n + 1 SEQUENCE

- Look at a simple sequence that has fascinated and foxed mathematicians for many years. They still cannot answer even quite simple questions about this.
- The "computational rule" for creating the sequence is to start from some given n, and to generate the next term of the sequence from n, either by halving n, (whenever n is even), or else by multiplying it by three and adding 1. The sequence terminates when n reaches 1.
- This Python snippet captures that algorithm:

```
n = 1027371
while n != 1:
print(n, end=", ")
if n % 2 == 0: # n is even
n = n // 2
else: # n is odd
n = n * 3 + 1
print(n, end=".\n")
```

- Notice first that the print function on line 4 has an extra argument end=", ". This tells the print function to follow the printed string with whatever the programmer chooses (in this case, a comma followed by a space), instead of ending the line. So each time something is printed in the loop, it is printed on the same output line, with the numbers separated by commas. The call to print(n, end=".\n") at line 9 after the loop terminates will then print the final value of n followed by a period and a newline character. (You'll cover the \n (newline character) later).
- The condition for continuing with this loop is n != 1, so the loop will continue running until it reaches its termination condition, (i.e. n == 1).
- Each time through the loop, the program outputs the value of n and then checks whether it is even or odd. If it is even, the value of n is divided by 2 using integer division. If it is odd, the value is replaced by n * 3 + 1.
- Since n sometimes increases and sometimes decreases, there is no obvious proof that n will ever reach 1, or that the program terminates. For some particular values of n, we can prove termination. For example, if the starting value is a power of two, then the value of n will be even each time through the loop until it reaches 1.
- We can find a small starting number that needs more than a hundred steps before it terminates. Particular values aside, the interesting question was first posed by a German mathematician called Lothar Collatz: the Collatz conjecture (also known as the 3n + 1 conjecture), is that this sequence terminates for all positive values of n. So far, no one has been able to prove it or disprove it! (A conjecture is a statement that might be true, but nobody knows for sure.)
- Think carefully about what would be needed for a proof or disproof of the conjecture "All positive integers will eventually converge to 1 using the Collatz rules". With fast computers we have been able to test every integer up to very large values, and so far, they have all eventually ended up at 1. But who knows? Perhaps there is some as-yet untested number which does not reduce to 1.
- We notice that if you don't stop when we reach 1, the sequence gets into its own cyclic loop: 1, 4, 2, 1, 4, 2, 1, 4 . . . So one possibility is that there might be other cycles that we just haven't found yet.
- Wikipedia has an informative article about the Collatz conjecture. The sequence also goes under other names (Hailstone sequence, Wonderous numbers, etc.), and we find out just how many integers have already been tested by computer, and found to converge!

CHOOSING BETWEEN FOR AND WHILE

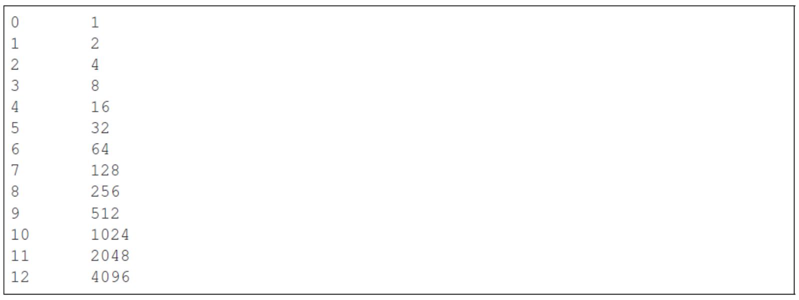
- Use a for loop if we know, before you start looping, the maximum number of times that we'll need to execute the body. For example, if we're traversing a list of elements, we know that the maximum number of loop iterations we can possibly need is "all the elements in the list". Or if we need to print the 12 times table, we know right away how many times the loop will need to run.
- So any problem like "iterate this weather model for 1000 cycles", or "search this list of words", "find all prime numbers up to 10000" suggest that a for loop is best.
- By contrast, if we are required to repeat some computation until some condition is met, and we cannot calculate in advance when (of if) this will happen, as we did in this 3n + 1 problem, you'll need a while loop.
- We call the first case definite iteration—we know ahead of time some definite bounds for what is needed. The latter case is called indefinite iteration — we're not sure how many iterations we'll need, we cannot even establish an upper bound!

TABLES

- One of the things loops are good for is generating tables. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other mathematical functions by hand. To make that easier, mathematics books contained long tables listing the values of these functions. Creating the tables was slow and boring, and they tended to be full of errors.
- When computers appeared on the scene, one of the initial reactions was, "This is great! We can use the computers to generate the tables, so there will be no errors." That turned out to be true (mostly) but shortsighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete.
- Well, almost. For some operations, computers use tables of values to get an approximate answer and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the Intel Pentium processor chip used to perform floating-point division.
- Although a log table is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and 2 raised to the power of that value in the right column:

```
for x in range(13): # Generate numbers 0 to 12
print(x, "\t", 2**x)
```

- The string "\t" represents a tab character. The backslash character in "\t" indicates the beginning of an escape sequence. Escape sequences are used to represent invisible characters like tabs and newlines. The sequence \n represents a newline.
- An escape sequence can appear anywhere in a string; in this example, the tab escape sequence is the only thing in the string. How do you think you represent a backslash in a string?
- As characters and strings are displayed on the screen, an invisible marker called the cursor keeps track of where the next character will go. After a print function, the cursor normally goes to the beginning of the next line.
- The tab character shifts the cursor to the right until it reaches one of the tab stops. Tabs are useful for making columns of text line up, as in the output of the previous program:



```
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
```

- Because of the tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

TWO-DIMENSIONAL TABLES

- A two-dimensional table is a table where you read the value at the intersection of a row and a column. A multiplication table is a good example. Let's say you want to print a multiplication table for the values from 1 to 6.
- A good way to start is to write a loop that prints the multiples of 2, all on one line:

```
for i in range(1, 7):
print(2 * i, end=" ")
print()
```

- Here we've used the range function, but made it start its sequence at 1. As the loop executes, the value of i changes from 1 to 6. When all the elements of the range have been assigned to i, the loop terminates. Each time through the loop, it displays the value of 2 * i, followed by three spaces.
- Again, the extra end=" " argument in the print function suppresses the newline, and uses three spaces instead. After the loop completes, the call to print at line 3 finishes the current line, and starts a new line.
- The output of the program is:

```
2    4    6    8    10    12
```

THE BREAK STATEMENT

- The break statement is used to immediately leave the body of its loop. The next statement to be executed is the first one after the body:

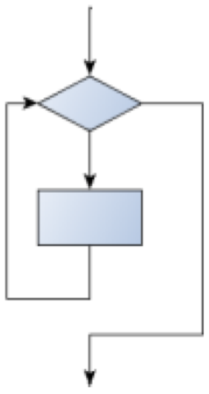
```
for i in [12, 16, 17, 24, 29]:
if i % 2 == 1: # If the number is odd
break # ... immediately exit the loop
print(i)
print("done")
```

This prints:

```
12
16
Done
```

THE PRE-TEST LOOP— STANDARD LOOP BEHAVIOUR

- for and while loops do their tests at the start, before executing any part of the body. They're called pre-test loops, because the test happens before (pre) the body. break and return are our tools for adapting this standard behaviour.



THE CONTINUE STATEMENT

- This is a control flow statement that causes the program to immediately skip the processing of the rest of the body of the loop, for the current iteration. But the loop still carries on running for its remaining iterations:

```
for i in [12, 16, 17, 24, 29, 30]:
if i % 2 == 1: # If the number is odd
continue # Don't process it
print(i)
print("done")
```

This prints:

```
12
16
24
30
done
```

PAIRED DATA

- Here it shows a more advanced way of representing our data. Making a pair of things in Python is as simple as putting them into parentheses, like this:

```
year_born = ("Paris Hilton", 1981)
```

- We can put many pairs into a list of pairs:

```
celebs = [("Brad Pitt", 1963), ("Jack Nicholson", 1937),
                                ("Justin Bieber", 1994)]
```

- Here is a quick sample of things we can do with structured data like this. First, print all the celebs:

```
print(celebs)
print(len(celebs))
```

```
[("Brad Pitt", 1963), ("Jack Nicholson", 1937), ("Justin Bieber", 1994)]
3
```

➢ Notice that the celebs list has just 3 elements, each of them pairs. Now we print the names of those celebrities born before 1980:

```
for name, year in celebs:
if year < 1980:
print(name)
```

```
Brad Pitt
Jack Nicholson
```

➢ This demonstrates something we have not seen yet in the for loop: instead of using a single loop control variable, we've used a pair of variable names, (name, year), instead. The loop is executed three times, once for each pair in the list, and on each iteration both the variables are assigned values from the pair of data that is being handled.

NESTED LOOPS FOR NESTED DATA

➢ Here we come up with an even more adventurous list of structured data. In this case, we have a list of students. Each student has a name which is paired up with another list of subjects that they are enrolled for:

```
students = [
        ("John", ["CompSci", "Physics"]),
        ("Vusi", ["Maths", "CompSci", "Stats"]),
        ("Jess", ["CompSci", "Accounting", "Economics", "Management"]),
        ("Sarah", ["InfSys", "Accounting", "Economics", "CommLaw"]),
        ("Zuki", ["Sociology", "Economics", "Law", "Stats", "Music"])]
```

➢ Here we've assigned a list of five elements to the variable students. Let's print out each student name, and the number of subjects they are enrolled for:

```
# Print all students with a count of their courses.
for name, subjects in students:
print(name, "takes", len(subjects), "courses")
```

Python agreeably responds with the following output:

```
John takes 2 courses
Vusi takes 3 courses
Jess takes 4 courses
Sarah takes 4 courses
Zuki takes 5 courses
```

➢ Now we'd like to ask how many students are taking CompSci. This needs a counter, and for each student we need a second loop that tests each of the subjects in turn:

```
# Count how many students are taking CompSci
counter = 0
for name, subjects in students:
for s in subjects: # A nested loop!
```

```
if s == "CompSci":
counter += 1
print("The number of students taking CompSci is", counter)
```

```
The number of students taking CompSci is 3
```

- A more concise of doing this would be the following:

```
counter = 0
for name, subjects in students:
if "CompSci" in subjects:
counter += 1
```

FUNCTIONS THAT REQUIRE ARGUMENTS

- Most functions require arguments: the arguments provide for generalization. For example, if we want to find the absolute value of a number, we have to indicate what the number is. Python has a built-in function for computing the absolute value:

```
>>> abs(5)
5
>>> abs(-5)
5
```

In this example, the arguments to the abs function are 5 and -5.

- Some functions take more than one argument. For example the built-in function pow takes two arguments, the base and the exponent. Inside the function, the values that are passed get assigned to variables called parameters.

```
>>> pow(2, 3)
8
>>> pow(7, 4)
2401
```

- Another built-in function that takes more than one argument is max.

```
>>> max(7, 11)
11
>>> max(4, 1, 17, 2, 12)
17
>>> max(3 * 11, 5**3, 512 - 9, 1024**0)
503
```

- max can be passed any number of arguments, separated by commas, and will return the largest value passed. The arguments can be either simple values or expressions. In the last example, 503 is returned, since it is larger than 33, 125, and 1.

FUNCTIONS THAT RETURN VALUES

- All the functions in the previous section return values. Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression.

```
biggest = max(3, 7, 2, 5)
x = abs(3 - 11) + 10
```

- So an important difference between these functions and one like draw_square is that draw_square was not executed because we wanted it to compute a value i.e, on the contrary, we wrote draw_square because we wanted it to execute a sequence of steps that caused the turtle to draw.
- A function that returns a value is called a fruitful function in this book. The opposite of a fruitful function is void function — one that is not executed for its resulting value, but is executed because it does something useful. (*Languages like Java, C#, C and C++ use the term "void function", other languages like Pascal call it a procedure.*)
- Even though void functions are not executed for their resulting value, Python always wants to return something. So, if the programmer doesn't arrange to return a value, Python will automatically return the value None.
- The standard formula for compound interest, which we'll now write as a fruitful function:

$$A = P\left(1 + \frac{r}{n}\right)^{nt}$$

Where,

- P = principal amount (initial investment)
- r = annual nominal interest rate (as a decimal)
- n = number of times the interest is compounded per year
- t = number of years

```
def final_amount(p, r, n, t):
"""
Apply the compound interest formula to p
to produce the final amount.
"""
a = p * (1 + r/n) ** (n*t)
return a # This is new, and makes the function fruitful.

# now that we have the function above, let us call it.
toInvest = float(input("How much do you want to invest?"))
fnl = final_amount(toInvest, 0.08, 12, 5)
print("At the end of the period you'll have", fnl)
```

- The return statement is followed an expression (a in this case). This expression will be evaluated and returned to the caller as the "fruit" of calling this function.
- We prompted the user for the principal amount. The type of toInvest is a string, but we need a number before we can work with it. Because it is money, and could have decimal places, we've used the float type converter function to parse the string and return a float.
- Notice how we entered the arguments for 8% interest, compounded 12 times per year, for 5 years.
- When we run this, we get the output

  At the end of the period you'll have 14898.457083

  This is a bit messy with all these decimal places, but remember that Python doesn't understand that we're working with money: it just does the calculation to the best of its ability, without rounding. Later we'll see how to format the string that is printed in such a way that it does get nicely rounded to two decimal places before printing.

- The line toInvest = float(input("How much do you want to invest?")) also shows yet another example of composition — we can call a function like float, and its arguments can be the results of other function calls (like input) that we've called along the way.

➢ Notice something else very important here. The name of the variable we pass as an argument — toInvest — has nothing to do with the name of the parameter—p. It is as if p = toInvest is executed when final_amount is called. It doesn't matter what the value was named in the caller, in final_amount its name is p.

➢ These short variable names are getting quite tricky, so perhaps we'd prefer one of these versions instead:

```
def final_amount_v2(principal_amount, nominal_percentage_rate,
num_times_per_year, years):
a = principal_amount * (1 + nominal_percentage_rate /
num_times_per_year) ** (num_times_per_year*years)
return a
def final_amount_v3(amount, rate, compounded, years):
a = amount * (1 + rate/compounded) ** (componded*years)
return a
def final_amount_v4(amount, rate, compounded, years):
""
The a in final_amount_v3 was a useless asignment.
We might as well skip it.
""
return amount * (1 + rate/compounded) ** (componded*years)
```

➢ They all do the same thing. Use your judgement to write code that can be best understood by other humans! Short variable names should generally be avoided, unless when short variables make more sense. This happens in particular with mathematical equations, where it's perfectly fine to use x, y, etc.