1. ORDER OF OPERATIONS
   a. () > ** > *., / > +,-
   b. Operations with the same precedence are evaluated left to right.
   c. Example: degrees / 2 * pi
   d. When in doubt, use parenthesis, and test things interactively in the interpreter.

2. FLOW OF EXECUTION
   a. Begins at the top. One by one towards the bottom.
   b. A call to a function is like a detour in the flow of execution.
   c. It executes everything inside the function, and then comes back to pick up where it left off.
   d. You can define many functions inside a script.
   e. Functions can call other functions.
   f. Show example

3. SCOPE OF VARIABLES IN A FUNCTION
   a. When it goes inside the function, the environment variable is new.
   b. When you create a variable inside a function, it is local. It only exists inside the function.
   c. Show example

4. WHY FUNCTIONS?
   a. Consolidate a group of statements. Easier to read and debug.
   b. Eliminates repetitive code. If you make a change, you only have to do it once.
   c. Allows to divide a long program into manageable parts. Solve one at a time.
   d. Well-designed functions can be useful for other programs - not just the one you are working on currently.

5. MODULUS OPERATOR
   a. Works on integers. Yields the remainder when the first operand is divided by the second.
   b. Example: remainder = 7 % 3
   c. Useful to check whether a number is divisible by another.
      i. If x % y is zero, then x is divisible by y.
   d. Also to extract digits from a number.
      i. x % 10 yields the right most digit of x.

6. BOOLEAN EXPRESSIONS
   a. An expression that is either true or false.
   b. Operators:
      i. == equal to? (common error to use single equal sign, which is an assignment)
      ii. != not equal to?
      iii. > greater than?
      iv. >= greater than or equal to?
      v. <= less than or equal to?
   c. Useful for testing.

7. LOGICAL OPERATORS
   a. Operators: and, or, not.
   b. Practice: What does n%2 == 0 or n%3 == 0 test?
8. CONDITIONAL EXECUTION
   a. So far we’ve only written linearly sequential programs.
   b. Just as in cooking, sometimes you need to check how things are going and change your actions. Taste, if salty, add balsamic vinegar, sometimes parsley helps.
   c. Any interesting program will need the ability to test conditions and change the behavior of the program accordingly.
   d. Example: Write a program that checks whether \( x \) is positive.
   e. If the condition is \( \text{true} \), then the indented statements get executed. If not, nothing happens.
   f. Execution resumes where the indentation goes back.
   g. There can be more than one statement in the conditional branch.

9. ALTERNATIVE EXECUTION
   a. One can also add an alternative execution.
   b. The \( \text{else} \) statement allows for a sequence of statements to be executed in the case that the condition is not true.
   c. Example: Program that checks whether \( x \) is positive or negative.
   d. Practice: Write a program that checks whether a number is even or odd (2 mins).

10. CHAINED CONDITIONALS
    a. Sometimes there are more than two possibilities.
    b. Else if, \( \text{elif} \), allows to test for additional conditions.
    c. Example: Program that checks whether \( x \) is less than \( y \), greater than \( y \), or equal to \( y \).

11. NESTED CONDITIONALS
    a. Conditionals can be nested within another.
    b. Example: Re-write the program that checks whether \( x \) is less, greater, or equal to \( y \).
    c. Practice: Write a program that receives three numbers, \( x \), \( y \), and \( z \), and returns the position of \( x \) relative to \( y \) and \( x \) (lowest, in the middle, or greatest) using only the \( \text{if}, \text{elif}, \text{and else} \) statements (no logical operators).

12. NESTED vs. LOGICAL OPERATOR
    a. Logical operators can often provide a way to simplify nested conditional statements.
    b. Example: Re-write the previous code to using the least amount of if, elif and else statement by using logical operators.

13. DEBUGGING TECHNIQUES
    a. We’ve been using print statements as the branches. Usually there will be more useful code inside the branches.
    b. However, using \textit{print} statements in different places of the code, particularly in the branches, allows us to get a good grip on the flow of execution.
    c. This is particularly useful when trying to debug our program.
    d. If you are not sure what’s wrong with your program, add print statements, and follow the execution through.
    e. That is one of the oldest and one of the best debugging techniques.
    f. When people come to me and ask me what’s wrong with their program. I will sit down next to them, and simply write print statements in their code. Then run it. Then try to figure out what’s happening. There’s really no more magical way of figuring out what’s wrong.
14. UPDATING VARIABLES
   a. x = 0
   b. x = x + 1

15. ITERATIONS: WHILE LOOPS
   a. Unlike us, computers are particularly good at doing repetitive tasks.
   b. The while statement tells the computer to repeat a block of statements until the condition is met.
   c. Example: Countdown example.

16. BRUTE FORCE ALGORITHMS
   a. Calculating the square root of a perfect square.
   b. Defensive programming.