**Variables, Data Types, Computations, and Assignment**

```
variable = expression
```

Variable is assigned the value on right, data type of variable becomes data type of value (variables may change type).

- \( N = 5 \)  
  Variable \( N \) is assigned an **integer**.
- \( N = 549583495830498534 \)  
  Variable \( N \) is assigned a **long integer**.
- \( N = 5.0 \)  
  Variable \( N \) is assigned a **float**.
- \( N = (5.0+7.0j) \)  
  Variable \( N \) is assigned a **complex**.
- \( N = "Hello World" \)  
  Variable \( N \) is assigned a **string** (can contain ')
- \( N = 'Hello World' \)  
  Variable \( N \) is assigned a **string** (can contain ')
- \( N = """Hello World"""" \)  
  Variable \( N \) is assigned a **string containing a line break** (can contain " or ").
- \( N = [1, 2.3, "Hello"] \)  
  Variable \( N \) is assigned a **list** (three items).
- \( N = (1, 2.3, "Hello") \)  
  Variable \( N \) is assigned a **tuple** (three items).
- \( N = True \)  
  Variable \( N \) is assigned a **bool** (Boolean).
- \( N = False \)  
  Variable \( N \) is assigned a **bool** (Boolean).

Variable names may be of essentially arbitrary length, must start with a letter, and may contain letters, digits, or the underscore. Case matters: variable `frog` is different from variable `Frog`, although variable names should be distinct even with respect to case to reduce errors. Names should reflect their usage: a variable to hold a total should be named `Total`, not `Q1`.

The break-over point between integers and long integers is implementation dependent. In JES, this point is at \( 2^{31}-1 \), or 2147483647. Any values less than or equal to that point are handled as integers directly in the processor chip; any greater values must be computed in software. Long integers have arbitrary precision, but computations using them take longer and longer times as the values get larger.

Complex numbers always use floats for the real and imaginary components. The imaginary component is indicated by the letter \( j \), not by \( i \) as you might expect from math. Python supports the basic math operations of \(+, -, *, \text{ and } /\) on complex numbers. For example, \((3+5j) + (9-2j) = (12+3j)\).

Floats are standard “double-precision floating-point” values. This is the same floating-point format as used in Microsoft Excel and a number of other numeric programs. The format allows 15-16 decimal digits of precision (significant figures), and a dynamic range of \( \pm 10^{308} \).

Python supports the basic math operations of \(+, -, *, \text{ and } /\) on integers and long integers. Results of integer computations too large to fit into an integer data type automatically default over to using long integers instead. Division on integers gives an integer result. For example, \( 7/2 = 3 \) (both operands are integers), but \( 7.0/2 = 7/2.0 = 7.0/2.0 = 3.5 \) (at least one operand is a float). Powers use a double-asterisk: \( 2**10 \) gives the result of computing \( 2^{10} \) (the result is an integer or long integer if both operands are integers, and are floats otherwise). There is also a \( // \) operator that performs integer division, but returns the most general data type of the operands. For example, \( 7//2 = 3 \) (both operands are integers), but \( 7.0//2 = 3.0 \) (the result, a float, has the fractional part discarded).

Boolean (bool) names `True` and `False` are aliases for integers 1 and 0, respectively.
Strings, lists, and tuples are all lists of various items; in strings the items are all characters, in lists and tuples the items may be of any arbitrary type, including other lists and tuples. For a variable named N (string, list, or tuple), the first item is at N[0], the second at N[1], up to the last at N[len(N)-1] (the Python len function returns the number of items in a string, list, or tuple). These items may be referenced with negative indices as well as positive, where N[-1] is the same as N[len(N)-1] (the last item), and N[-len(N)] is the same as N[0] (the first item). Any index greater than or equal to len(N) or less than -len(N) causes a run-time error.

Strings and tuples are immutable, meaning that component items may not be changed. Lists are mutable, allowing individual items of the list to be changed. For example, the assignment statement N[0] = "X" is illegal for strings and tuples, but perfectly legal for lists. However, the + (plus) operator works for strings, lists, and tuples as a concatenation operator: two strings may be added, two lists may be added, or two tuples may be added (but no mixing of types), and in all three cases the elements of the second operand are concatenated (joined) onto the end of the first operand. For example, "dog"+"house" = "doghouse", [3,7]+[9,"dog",1] = [3,7,9,"dog",1], and (3,7)+(9,"dog",1) = (3,7,9,"dog",1). Empty strings are denoted by "" or ' ', empty lists by [], and empty tuples by ().

Simple Program Statement structures

Comments start with #. The # symbol can appear anywhere on a line; at the beginning of the line to make the entire line a comment, or after a statement on that line.

Statements in Python are (almost) always one instruction per line of text. Long Python instructions may be broken across lines of text if on every line except the last the terminating character is the \ character.

Blocks of code that are part of an enclosing structure (if, while, def, etc.) are indented relative to that structure, and all lines that are part of the same block must be indented by the same amount. It is a good idea to be consistent in indentation; four spaces per indentation level is recommended.

Functions

```python
def FunctionName (parameter, parameter, ...):
    # statements
    # statements
    return value

def FunctionName ():
    # statements
    # statements
    return
```

Statements are always indented relative to the def statement.

The parameters in a def statement are called formal parameters and act as placeholders within the function for the values passed in. Those variable names are local to the function, and are not visible outside the body of the function. Parameter values are assigned when the function is called with actual parameters, which may be variables of the same or different names, or constants. When a function returns, it returns to the next statement after the one that called it.

Parameters are call-by-value where changes to those variables inside the function aren’t seen outside the function. When parameters are pointers to objects (such as pictures, pixels, colors, etc.), operations
can change the referenced object being pointed to, and those changes are seen by the calling routine. The parameter list is allowed to be empty if nothing is to be communicated into the function.

A return value is optional, and is used when a function has to send information back to the routine that called it. The return value is always a single item, but may be a string, list, or a tuple to return several values at once. The only other way to communicate values into or out of a function is through global variables, the use of which is not recommended except in unusual circumstances. The return statement may appear anywhere inside a function, not just at the end. If a function ends without a return statement (not recommended), the function automatically returns at that point.

Parameters may also have a default value specified in the def statement, allowing that parameter to be omitted when the function is called. If a value is specified when the function is called, the parameter uses the specified value instead of the default value. Default values are defined on parameters from right-to-left only (that is, if you have a parameter with a default value, all parameters to the right of that one must also have default values – you can’t skip any – and omitting one in the function call means you must omit all those to the right as well).

```
def FunctionName (parameter, parameter, parameter=defaultvalue):
    # statements
    # statements
    return
```

Functions may be passed as parameters to other functions. Functions may call themselves (recursion). Local functions may be defined inside a function, and are visible only to that function. These are advanced techniques.

**Simple Statement structures**

All programs are formed from sequences, selections, and iterations. A sequence is a list of statements that must be performed in the order they appear in the program, and all must have the same indentation level in Python. A selection is a way that the program can decide which of one or two separate paths is to be followed, in Python is denoted by the if and (optional) else statements. Multiple if statements in a long chain can be simplified with the else-if) statement. An iteration indicates a loop, where the same statements are executed many times. The most general form of an iteration in Python is the while loop, along with a more efficient form called the for loop (a for loop can always be written as an equivalent while loop, but not all while loops can be written as for loops).

As programs are being written, a programmer may recognize that a structure (such as an if or while) is needed, but the body of that structure must be written at a much later time. In order to have the incomplete structure remain acceptable to Python, the body of that structure can be replaced with the pass statement. The pass statement takes no action on its own. It will be replaced with the correct code when the programmer gets around to writing it.
The if Statement

An if statement is a way for the program to decide whether or not an action is to be performed. The keyword if is always followed by a true-or-false question and a colon. The statements on the lines following are executed only in the case where the question is true. Because those statements “belong” to the if statement, they are indented one level. In the following example, the indented statements are executed if the question is true, but are skipped (and nothing is done) if the question is false.

```python
if (question):
    # statements
    # statements
```

If the question is true do the following statements. Use pass if there is nothing to do right now.

In most circumstances, there is an alternate path to follow if the question is false. This is shown in the following example, where the statements after the if are performed if the question is true, but the statements after the else are performed if the question is false.

```python
if (question):
    # statements
    # statements
else:
    # statements
    # statements
```

If the question is true do the following statements. Use pass if there is nothing to do right now.

If the question is not true, statements after the else are executed. Use pass if there is nothing to do right now.

If the else section contains only the pass statement, the entire else section can be omitted.

In many programs it is necessary to choose one of a number of actions based on a series of questions. Using only the if-else combination, the resulting code would look as follows:

```python
if (question1):
    # statements
    # statements
else:
    if (question2):
        # statements
        # statements
    else:
        if (question3):
            # statements
            # statements
        else:
            # statements
            # statements
```

If the question is true do the following statements.

If the first if is false, ask a follow-up question.

If the second if is false, ask another follow-up question.

If none of the questions are true, statements after the last else are executed.

It is obvious that asking a large number of questions in a row will result in a long, heavily-indented program. There is, however, a cleaner way in Python to perform the same actions.
In the following example, all cases where else is followed by if have been collapsed into the special keyword elif, and all blocks of code controlled by the if, elif, and else statements have the same indentation level. Very long sequences are therefore no more complicated than short ones.

```python
if (question1):
    # statements
    # statements
elif (question2):
    # statements
    # statements
elif (question3):
    # statements
    # statements
else:
    # statements
    # statements
```

If the question is true do the following statements. Use pass if there is nothing to do right now.

If the first if is false, an elif can ask a follow-up question. (An elif is a contraction of else-if)

If the first elif is false, a second can ask another follow-up question.

If none of the questions are true, statements after the else are executed.

**The while loop**

The if-elif-else statements allow a program to choose one action from many possibilities, but that action is only performed once. To make an action be performed many times requires a loop. There are two styles of loops; the first is called the while loop. The following example shows a typical loop, wherein the semantics are “while the question is true do the statements over and over”. Like an if, the question is tested to see whether or not it is true, and if so the statements are executed. Unlike an if, however, when the statements have been executed the program loops back and asks the question again. If true, the statements are executed again. This process continues until the first time that the question is false, at which time the code beyond the bottom of the loop is executed.

```python
while (question):
    # statements
    # statements
```

Test to see if the question is true, if so do the statements, then go back and ask the question again. Loop while the question remains true.

**Boolean Questions**

The “questions” used in if, elif, and while statements are almost always tests that return either True or False as their results. Usually, this entails comparing two items (two integers, two strings, etc.) to see how one relates to the other by placing a special operator between those two items:

Are two things equal?  
==
Are two things not equal?  
!= or <>
Is the first less than the second?  
<
Is the first greater than the second?  
>
Is the first less than or equal to the second?  
<=
Is the first greater than or equal to the second?  
>=
Is the first item in a string, list, or tuple?  
in
For example, to test if variable \( N \) equals zero the question is written as \((N == 0)\), to test if variable \( I \) is less than 10 the question is written as \((I < 10)\), etc. Here are a couple of (meaningless) examples of how questions are used in context:

\[
\text{if } (N == 0): \\
\quad S = "Zero" \\
\text{else:} \\
\quad S = "Not Zero"
\]

\[
\text{while } (I < 10): \\
\quad \text{Total} = \text{Total} + I \\
\quad I = I + 1
\]

For the detail-minded: the result of any question is actually the integer value 1 if it is true and the integer value 0 if it is false. (The Python word \texttt{True} is simply an alias for the integer 1, and \texttt{False} is an alias for the integer 0.) In \texttt{if}, \texttt{elif}, and \texttt{while} statements, Python actually tests for whether or not the question is zero; any non-zero value is considered to be true, and only zero itself is considered to be false. This means that the statement \texttt{if (X)} is effectively the same as \texttt{if (X != 0)}: (although beginning programmers should avoid this as it can lead to subtle errors). Sophisticated programmers can use questions directly in computations as well. The following two examples have the same computational result, namely to add 1 to \( N \) if \( X \) is greater than zero:

\[
\text{if } (X > 0): \\
\quad N = N + (X > 0)
\]

\[
\text{The \texttt{range} function}
\]

The second type of loop is called the \texttt{for} loop, but before that can be described we need to cover the \texttt{range} function. There are three ways of using the \texttt{range} function, with one, two, or three integer parameters, and the result is always a list:

**The \texttt{range} function with 1 integer parameter:** The result is a list of integers from 0 up to one less than the parameter. The parameters must be positive. For example, \texttt{range(5)} returns \([0, 1, 2, 3, 4]\), \texttt{range(10)} returns \([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]\), and \texttt{range(-10)} returns \([\text{empty list}]\).

**The \texttt{range} function with 2 integer parameters:** The result is a list of integers from the first parameter up to one less than the second parameter. The parameters may be either positive or negative, but the second must be greater than the first. For example, \texttt{range(3, 10)} returns \([3, 4, 5, 6, 7, 8, 9]\), \texttt{range(-1, 2)} returns \([-1, 0, 1]\), and \texttt{range(2, -1)} returns \([\text{empty list}]\).

**The \texttt{range} function with 3 integer parameters:** The result is a list of integers from the first parameter through but not including the second parameter, in steps of the third parameter. The parameters may be positive or negative, but the second must be greater than the first if the third is positive, and the second must be less than the first if the third is negative. The third parameter may not be exactly zero. For example, \texttt{range(3, 10, 2)} returns \([3, 5, 7, 9]\), and \texttt{range(10, 3, -2)} returns \([10, 8, 6, 4]\).
The **for** loop

The second type of loop is the **for** loop. Like the **while** loop, the **for** loop can execute a block of statements many times, but unlike the **while** loop the number of times is always known at the time the **for** loop starts up. The syntax is always **for** **variable** **in** **list**: where on each pass through the loop the variable takes on one of the values from the list. Many times the list is generated by the **range** function. Here are some examples:

```python
for variable in list:
    # statements
    # statements

for I in [6,2,10]:
    # statements
    # statements

for I in range(10):
    # statements
    # statements

for I in range(10,15):
    # statements
    # statements

for I in range(10,20,3):
    # statements
    # statements
```

Do the statements for every value in the list. Usually combined with the **range** function to build lists.

Do the statements 3 times, with I taking on successively the values from [6,2,10].

Do the statements 10 times, with I taking on successively the values from [0,1,2,3,4,5,6,7,8,9].

Do the statements 5 times, with I taking on successively the values from [10,11,12,13,14].

Do the statements 4 times, with I taking on successively the values from [10,13,16,19].

Most **for** loops may be written as **while** loops (not vice-versa, however), but a **for** loop is much more efficient than its equivalent **while** loop. Here is an example. Notice how much more code needs to be written for the **while** loop than for the **for** loop:

```python
for I in range(3,10,2):
    # statements
    # statements

I = 3
while (I < 10):
    # statements
    # statements
    I = I + 2
```
In the JES environment, the `getPixels` function returns a list of pixel-pointers, which may be used in a `for` loop identically to the examples above. This example scans through every pixel in an image and allows the body of the loop to perform some processing on those pixels one at a time:

```python
for PX in getPixels(Canvas):
    # statements
    # statements
    repaint(Canvas)
```

Do the statements as many times as there are pixels in the Canvas, where `PX` takes on successive pixel pointers. `repaint(Canvas)` Repaint after everything is finished.

Just for completeness, here is a variation that uses the width and height of the image to process pixels row-by-row, updating the image on screen after each row is complete:

```python
for Y in range(getHeight(Canvas)):
    for X in range(getWidth(Canvas)):
        PX = getPixel(Canvas,X,Y)
        # statements
        # statements
        repaint(Canvas)
```

Step through every row of a Canvas Step through every column of each row Pick up the pixel at location <X,Y>. Process the pixel. `repaint(Canvas)` Repaint after every line.

**Python shortcuts**

**Math Shortcuts:** Many statements assign a value to a variable based on a computation involving that same variable. For example, adding 1 to `N` is written as `N = N + 1`, and multiplying `I` into `F` is written as `F = F * I`. These types of computations may be optimized by using special operators formed by joining the math operation to the equals sign. For example, `N = N + 1` may be written as `N += 1`, and `F = F * I` may be written as `F *= I` instead. These forms are slightly more computationally efficient than the spelled-out versions.

**Statement Shortcuts:** In most structural statements that end in a colon (`if`, `elif`, `else`, `while`, or `for`, but not `def`), if there is only a single simple instruction controlled by that statement, that one instruction may be placed on the same line as the statement which controls it. This has the effect of shortening the program, but it only applies when there is a single statement to join. Here are some examples of code without and with the optimization:

```python
if (X < 0):
    Answer = "Negative"
elif (X > 0):
    Answer = "Positive"
else:
    Answer = "Zero"
```

```python
if (X < 0):
    Answer = "Negative"
else:
    Answer = "Zero"
```

```python
for I in range(10):
    X = X * I
```

```python
for I in range(10):
    X = X * I
```

```python
for I in range(10):
    X = X * I
```