An interactive session in IDLE, with notes

You can type Python expressions and statements at the prompt in IDLE, and have them evaluated immediately. This is Python in interactive mode. It’s not the way we usually use the language, but it will help understand some of the grammar of Python.

CONSTANTS, OPERATORS, EXPRESSIONS

We combine numbers with the operation symbols +, -, *, /, and, at times parentheses. The resulting combinations are EXPRESSIONS.

```python
>>> 37+23
60
>>> 37*23
851
>>> 23-37
-14
>>> 37*23+5
856
>>> 37*(23+5)
1036
```

In the absence of parentheses, Python follows some fairly standard rules for parsing ambiguous expressions like '37*23+5'. Here the rule is usually described as 'multiplication takes precedence over addition'. Would the result have been any different if we had entered 5+37*23?

If you form an expression incorrectly, Python lets you know about it. The one below has mismatched parentheses:

```python
>>> 37*(23+5))
SyntaxError: invalid syntax
```

We haven't tried division yet.

```python
>>> 37/23
1
```

You may be surprised by this result. The constants '37' and '23' denote integers, and in this instance / denotes integer division, which just gives you the integer quotient with no fractional part, and throws away the remainder.
Do you want to also know the remainder? Python's got an operator for that, too: It's denoted %, but we usually call it mod, which is how it is denoted in mathematical writing.

```python
>>> 37%23
14

>>> 37**2
1369
```

One more operator: in Python, '**' denotes exponentiation: In standard mathematical notation, the expression below would be written $37^2$. You should be aware that, while '*' is pretty standard as a symbol for multiplication in computer languages, there is not the same kind of consensus on how to denote exponentiation. For example, in Excel spreadsheets, the same expression would be written $37^2$, which means something completely different in Python.

**TYPES**

Does what we said above about integer division mean that we can't use Python to divide 37 by 23 and get the 'normal' answer, which is somewhere north of one-and-a-half? Of course not! Here is one way to do it:

```python
>>> 37.0/23
1.608695652173913
```

Here is something important. In mathematics, 37.0 and 37 denote the same number, but in Python these constants represent different TYPES of numbers. The Python function 'type' will tell you what these types are called.

```python
>>> type(37)
<type 'int'>

>>> type(37.0)
<type 'float'>
```

The constant 37.0 is a **floating-point number**; it is different from the integer 37, and in fact these two values are represented in the computer in completely different ways. When the Python interpreter encounters an expression like 37.0/23, containing both a float and an int, it converts 23 to the corresponding floating-point number and performs floating-point division. (The result cannot be
Let's evaluate an expression that has a very large value

```python
>>> 37**37
105513495577783414078330085995832946127396083370199442517L
```

When the arguments are given as integers, Python evaluates the expression EXACTLY. The result is bigger than the largest available size for the int data type, so a special 'long integer' type is used instead:

```python
>>> type(37**37)
<type 'long'>
```

Most of the time, you don't have to worry about the distinction between long and int types; things usually take care of themselves. This is in contrast to the distinction between int and float, which you must pay attention to in order to avoid certain kinds of errors.

If we perform the same operation in floating-point arithmetic, we get a different-looking answer:

```python
>>> 37.0**37
1.055513495577783e+58
```

In standard mathematical notation, we would write this as \(1.055513495577783 \times 10^{58}\) Its type is still float.

```python
>>> type(37.0**37)
<type 'float'>
```

**VARIABLES AND ASSIGNMENT**

```python
>>> x=5
```

The assignment statement above attaches the name x to an object of type int that has value 5. We can now ask for the value of the variable x:

```python
>>> x
5
```
We can perform further calculations using the value of this variable, and assign the result to a new variable:

```python
>>> y=2*(x+3)
```

```python
>>> y
16
```

```python
>>> type(y)
<type 'int'>
```

```python
>>> y=y+5.1
```

We can have the same variable on the left and right-hand side of an assignment statement. This looks like an (impossible) equation, but assignment statements are not equations---they are instructions to the computer to do something. The result of the above assignment is to create a new object (of type float) and associate the name of the variable y to this object.

```python
>>> y
21.1
```

```python
>>> type(y)
<type 'float'>
```

Python has a square root function, called `sqrt`. Let's try it out:

```python
>>> sqrt(2)
```

```
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    sqrt(2)
NameError: name 'sqrt' is not defined
```

The reason Python did not recognize this name is because the `sqrt` function lives in another library, called `math`. Let's try again.

```python
>>> math.sqrt(2)
```
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    math.sqrt(2)
NameError: name 'math' is not defined

Python is still unhappy, because we cannot use the functions in the math library until we direct Python to import the library.

>>> import math

>>> math.sqrt(2)

1.4142135623730951

What if we tried to call a variable 'import'?

>>> import=3

SyntaxError: invalid syntax

>>> important=3

>>> important

3

There are a few rules for naming variables, but they don't have to all be stupid one-letter names like x and y. However a variable cannot have the same name as a Python reserved word. The above computation shows what happens when we try to treat the reserved word import as a variable.

STRINGS

The three Python types we have seen so far all represent numbers. There are many other types available in the language. We'll have a lot more to say about strings later on, but here are a few basics. String support a limited set of operations: you can add two strings, and 'multiply' a string by a positive integer.

>>> greeting1='hello'
>>> type(greeting1)
<type 'str'>
>>> greeting2='goodbye'
>>> greeting1+greeting2
'hellogoodbye'

>>> greeting1*3  
'hellohellohello'

>>> greeting1*greeting2

Traceback (most recent call last):
 File "<pyshell#45>" , line 1 , in <module>  
greeting1*greeting2
TypeError: can't multiply sequence by non-int of type 'str'

If you multiply a string by 0, the result is the VERY useful empty string.

>>> greeting1*0
''

TYPE CONVERSION

Python has built-in functions called int, float, and str that can be used to convert between types. For example, we could have solved our problem above about calculating the floating-point quotient of 37 by 23 by typing

>>> float(37)/23
1.608695652173913

We can also extract the integer part of a floating point number.

>>> int(4.72)
4

This is especially useful for converting the string representation of a number to its corresponding numerical value.

>>> int('3')
3
>>> int('3 ')  
3
>>> float('3.785')
3.785
>>> float('3')
3.0

Be careful---not everything you type here is legal, even if it appears to make sense.
>>> int('3.0')

Traceback (most recent call last):
  File "<pyshell#20>" , line 1, in <module>
    int('3.0')
ValueError: invalid literal for int() with base 10: '3.0'