> *Our galaxy itself contains a hundred million stars;*
> *It's a hundred thousand light-years side to side;*
> *It bulges in the middle sixteen thousand light-years thick,*
> *But out by us it's just three thousand light-years wide.*

Arithmetic, Basic Types and Variables

Python Arithmetic and Integers

Python understands arithmetical expressions and so the interpreter can be used as a calculator. For example:

```
>>> 2 + 3 * 4
14
>>> (2 + 3) * 4
20
>>> 10 - 4 - 3
3
>>> 10 - (4 - 3)
9
>>> -3 * 4
-12
>>> 2 ** 3
8
>>> 2 * 3 ** 2
18
>>> (2 * 3) ** 2
36
>>> 7 // 3
2
```

(Notice how ** is the Python operator for exponentiation.)

There are a few things worth pointing out about these examples.

Firstly, the Python interpreter reads user input as strings – i.e. sequences of characters. So for example, the first user input is read in as the string "2 + 3 * 4". This is treated by the interpreter as a piece of syntax – in this case valid syntax that represents an arithmetic expression. The interpreter then uses the semantics to convert from a string representing an arithmetic expression into the actual expression. So the substring "2" is converted to its semantics (meaning) – i.e. the number 2. The substring "+" is converted to its meaning – i.e. the addition *operator*. Once the interpreter converts from the syntax to the semantics, the interpreter then evaluates the expression, displaying the result.

This process of converting from syntax to semantics is called **parsing**.

Secondly, the relationship between the arithmetical operators is what humans normally expect – for example, multiplication takes precedence over addition. When looking at expressions involving operators, we need to consider both **precedence** and **associativity**.

Precedence of operators describes how tightly the operators bind to the arguments. For example, exponentiation binds more tightly than multiplication or division, which in turn bind more tightly than addition or subtraction. Precedence dictates the order of operator application (i.e. exponentiation is done first, etc.)

Associativity describes the order of application when the same operator appears in a sequence – for example `10 - 4 - 3`. All the arithmetical operators are left-associative – i.e. the evaluation is from left to right. Just like in mathematics, we can use brackets when the desired expression requires the operators to be evaluated in a different order.

Finally, dividing one integer by another, using the `//` divide operator (known as the div operator), produces an integer – the fractional part of the answer is discarded.

These examples are about integer arithmetic – integers form a built-in type in Python.

```
>>> type(3)
<class 'int'>
```

Note how the abbreviation for integers in Python is **int**. Also, note that `type` is a built-in Python function that takes an object (in this case the number 3) and returns the type of the object. We will see plenty of examples of Python functions in following sections.

Floats

There is another number type that most programming languages support – **floats**. Unlike integers, which can only store whole numbers, floats can store fractional parts as well. Below are some examples involving floats in Python.

```
>>> 7/4
1.75
>>> 7//4
1
>>> -7//4
-2
>>> 7.9//3
2.0
>>> 2.0**60
1.152921504606847e+18
>>> 0.5**60
8.6736173798840355e-19
>>> 2e3
2000.0
>>> type(2e3)
<class 'float'>
>>> int(2.3)
2
>>> float(3)
3.0
>>> 5/6
0.8333333333333334
>>> -5/6
-0.8333333333333334
```

The syntax for floats is a sequence of characters representing digits, optionally containing the decimal point character and also optionally a trailing 'e' and another sequence of digit characters. The character sequence after the 'e' represents the power of 10 by which to multiply the first part of the number. This is called **scientific notation**.

The first example shows that dividing two integers with the `/` operator gives a float result of the division. The single `/` division is known as **float division** and will always result in a float.

If an integer result is desired, the `//` operator must be used. This is the **integer division** operator and will divide the two numbers as if they are integers (performs the mathematical operation **div**). **Note** that this will always round **down** towards negative infinity, as shown in the third example. If one of the numbers is a float, it turns both numbers into their integer form and then performs the integer division but returns a float answer still, as seen in the third example.

The last two examples highlights one very important aspect of floats – **they are approximations to numbers**. Not all decimal numbers can be accurately represented by the computer in its internal representation. This is because floats occupy a fixed chunk of memory. The amount of memory used to store a float constrains the precision of the numbers that it can represent.

Variables and Assignments

Most calculators allow results to be stored away in memory and later retrieved in order to carry out complex calculations. This can be done in Python by using **variables**. As we will soon see, variables are not just for numerical calculations but can be used to store any information for later use.

The valid syntax for variables is any string starting with an alphabetic or underscore character ('a'-'z' or 'A'- 'Z' or '_') followed by a sequence of alphanumeric characters (alphabetic + '0'-'9') and '_'. Python uses special **keywords** itself and these cannot be used for any other purpose (for example, as variable names). The Python keywords are listed in the table below. A way of knowing that a word is a keyword is that it will appear in a different colour in IDLE.

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

In later sections, we will see that it is important to choose a good name for variables that accurately shows what the value is meant to represent. There is a convention as well in Python of using lowercase letters in variable names and underscores to separate words, which is adhered to in these notes. There is not anything stopping us ignoring the points in this paragraph, but it becomes an unnecessary annoyance for other people reading our code.

Below are some examples of the use of variables.

```
>>> num1 = 2
>>> num1
2
>>> 12**num1
144
>>> num2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'num2' is not defined
>>> num3 = num1**2
>>> num3
4
```

The first example above shows an **assignment** – the variable `num1` is assigned the value `2`. The semantics of the **assignment statement** is: evaluate (i.e. work out the value of) the right hand expression and associate the variable on the left hand side of the statement with that value. The left hand side of an assignment statement must be a variable (at least for now – later we will see other possibilities). If the variable already has a value, that value is overwritten by the value on the right hand side of the statement. Note that the assignment statement (when evaluated) has no value – the interpreter does not display a value. Assignments are used for their **side-effect** – the association of a variable with a value.

The second example shows what happens when we ask the interpreter to evaluate `num1`. The result is the value associated with the variable. The third example extends this by putting `num1` inside an arithmetical expression. The fourth example shows what happens when we try to get the value of the variable `num2`, to which we have not yet given a value. This is known as an **exception** and will appear whenever something is typed that Python cannot understand or evaluate. The final example shows how variables can appear on both sides of an assignment (in this case, a variable called `num3` is assigned the value worked out by evaluating `num1**2`; the value of `num1` is 2, so `num1**2` is `4` and this is the value given to `num3`.

Strings

To finish this section we introduce the **string** type, which is used to represent text, and we give some examples of type conversion and printing.

```
>>> s = "Spam "
>>> 3 * s
'Spam Spam Spam '
>>> 2 + s
Traceback (most recent call last):
  File "<stdin>", line 1, in
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> type('')
<class 'str'>
>>> int("42")
42
>>> str(42)
'42'
>>> print(s + 42)
Traceback (most recent call last):
  File "<stdin>", line 1, in
TypeError: cannot concatenate 'str' and 'int' objects
>>> print(s + str(42))
Spam 42
```

The syntax for strings is a double or single quote, followed by a sequence of characters, followed by a matching double or single quote. Notice how the space character is part of the string assigned in the variable `s`, as it is contained within the double quotes. The string `s`, therefore, is *five* characters in length. Notice carefully how this affects the statements executed after it.

As the second example shows, the multiplication operator is overloaded and generates a string that is the number copies of the original string.

Type conversion is used to change between a string representing a number and the number itself.

The built-in `print` function takes a string argument and displays the result to the user. Note, that in this example, `str` converts `42` into a string and then the addition operator concatenates (i.e. joins) the two strings together and the result is displayed to the user – another example of overloading.