1. Python is just the programming language for you.
2. Python is simple to use, but it is a real programming language,offering much more structure and support for large programs than shell scripts or batch files can offer.
3. Python also offers much more error checking than C, and, being a very-high-level language, it has high-level data types built in, such as flexible arrays and dictio- naries.
4. Python allows you to split your program into modules that can be reused in other Python programs.
5. It comes with a large collection of standard modules that you can use as the basis of your programs — or as examples to start learning to program in Python.
6. Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary.
7. The interpreter can be used interactively, which makes it easy to experiment with features of the language.
8. It is also a handy desk calculator.
9. Python enables programs to be written compactly and readably.
10. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:
    10.1. the high-level data types allow you to express complex operations in a single statement;
    10.2. statement grouping is done by indentation instead of beginning and ending brackets;
    10.3. no variable or argument declarations are necessary.
11. Python is extensible: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form.
12. By the way, the language is named after the BBC show “Monty Python’s Flying Circus” and has nothing to do with reptiles.

AN INFORMAL INTRODUCTION TO PYTHON

1. Comments in Python start with the hash character, #, and extend to the end of the physical line.
2. A comment may appear at the start of a line or following whitespace or code, but not within a string literal.

Using Python as a Calculator

Numbers

```
The interpreter acts as a simple calculator: you can type an expression
at it and it will write the value.

Expression syntax is straightforward: the operators +, -, * and / work
just like in most other languages (for example, Pascal or  C);
parentheses (()) can be used for grouping. For example:
```

```
2+2
```

```
4
```

```
50-5*6
```

```
20
```

```
(50-5*6)/4
```

```
5.0
```

```
8/5         #    division    always    returns    a    floating    point    number
```

```
1.6
```

The integer numbers (e.g. 2, 4, 20) have type int, the ones with a fractional part (e.g. 5.0, 1.6) have type float.

Division (/) always returns a float. To do floor division and get an integer result you can use the // operator; to calculate the remainder you can use %:

```
17/3         #    classic    division    returns    a    float
```

```
5.666666666666667
```

```
17//3         #    floor    division    discards    the    fractional    part
```

```
5
```

```
17%3         #    the    %    operator    returns    the    remainder    of    the    division
```

```
2
```

```
5*3+2         #    floored    quotient    *    divisor    +    remainder
```

```
17
```

With Python, it is possible to use the ** operator to calculate powers

```
5**2         #    5    squared
```

```
25
```

```
2 ** 7 # 2 to the power of 7
```

```
128
```

The equal sign (=) is used to assign a value to a variable.

```
width=20
height = 5 * 9
width * height
```

```
900
```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```
n
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-13-ab0680a89434> in <module>
----> 1 n

NameError: name 'n' is not defined
```

SEARCH STACK OVERFLOW

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
4*3.75-1
```

```
14.0
```

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
tax=12.5/100
price=100.50
price*tax
```

```
12.5625
```

```
price+_
```

```
113.0625
```

```
round(_,2)
```

```
113.06
```

This variable should be treated as read-only by the user. Don’t explicitly assign a value to it

In addition to int and float, Python supports other types of numbers,
such as Decimal and Fraction.

Python also has built-in support for complex numbers, and uses the j or
J suffix to indicate the imaginary part (e.g. 3+5j).

```
complex (3.)   #Constructing a complex object from a float generates a complex number with the imaginary part equal to zero.
```

```
(3+0j)
```

```
#To generate a pure imaginary number, you have to explicitly pass two numbers to complex with the first, real part,
# equal to zero.
complex (0. ,3.)
```

```
3j
```

Typing a number at the Python shell prompt simply echoes the
number back to you:

```
5
```

```
5
```

```
5.
```

```
5.0
```

```
0.10
```

```
0.1
```

```
0.0001
```

```
0.0001
```

```
0.0000999
```

```
9.99e-05
```

```
A number of one type can be created from a number of another type with
the relevant constructor:
```

```
float (5)
```

```
5.0
```

```
int (5.2)
```

```
5
```

```
#Note that a positive floating-point number is rounded down in casting it into an integer;
#more generally, int rounds towards zero: int(-1.4) would yield -1
```

```
int (5.9)
```

```
5
```

```
complex(1,2)
```

```
(1+2j)
```

```
#Constructing a complex object from a float generates a complex number with the imaginary part equal to zero.
complex(9.)
```

```
(9+0j)
```

```
#To generate a pure imaginary number, you have to explicitly pass two numbers to
# complex with the first, real part, equal to zero.
complex(0.,6.)
```

```
6j
```

Strings

```
Besides numbers, Python can also manipulate strings, which can be
expressed in several ways.

They can be enclosed in single quotes ('...') or double quotes ("...
")  with the same result2 .  \  can be used to escape quotes:
```

```
'spam eggs'         #    single    quotes
```

```
'spam eggs'
```

```
'doesn\'t'         #    use    \'    to    escape    the    single    quote...
```

```
'doesn't'
```

```
"doesn't"         #    ...or    use    double    quotes    instead
```

```
'doesn't'
```

```
'"Yes,"    they    said.'
```

```
'"Yes,"    they    said.'
```

```
"\"Yes,\"    they    said."
```

```
'"Yes,"    they    said.'
```

```
'"Isn\'t,"    they    said.'
```

```
'"Isn\'t,"    they    said.'
```

The print() function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
'"Isn\'t,"    they    said.'
```

```
'"Isn\'t,"    they    said.'
```

```
print('"Isn\'t,"    they    said.')
```

```
"Isn't,"    they    said.
```

```
s='First    line.\nSecond    line.'         #    \n    means    newline
s
```

```
'First    line.\nSecond    line.'
```

```
print(s)
```

```
First    line.
Second    line.
```

If you don't want characters prefaced by \ to be interpreted as special characters, you can use raw strings by adding an r before the first quote:

```
print('C:\some\name')         #    here    \n    means    newline!
```

```
C:\some
ame
```

```
print(r'C:\some\name')         #    note    the    r    before    the    quote
```

```
C:\some\name
```

```
#    There    is    one    subtle    aspect    to    raw    strings:    a    raw    string    may    not    end    in    an    odd    number    of    \    characters
print(r'C:\some\name\')
```

```
  File "<ipython-input-49-4def629fd8bd>", line 2
    print(r'C:\some\name\')
                         ^
SyntaxError: EOL while scanning string literal
```

SEARCH STACK OVERFLOW

String literals can span multiple lines.

One way is using triple-quotes: """...""" or '''...'''.

End of lines are automatically included in the string, but it's possible to prevent this by adding a \ at the end of the line. The following example:

```
print("""\
Usage: thingy [OPTIONS]
-h                    Display    this    usage    message
-H    hostname        Hostname    to    connect    to
""")
```

```
Usage: thingy [OPTIONS]
-h                    Display    this    usage    message
-H    hostname        Hostname    to    connect    to
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
#    3    times    'un',    followed    by    'ium'
```

```
3*'In'+'dia'
```

```
'InInIndia'
```

Two or more string literals (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```
'Py'    'thon'
```

```
'Python'
```

```
'Py''thon'
```

```
'Python'
```

```
'Py' 'thon'
```

```
'Python'
```

This feature is particularly useful when you want to break long strings:

```
text    =    ('Put    several    strings    within    parentheses    '
'to    have    them    joined    together.')
text
```

```
'Put    several    strings    within    parentheses    to    have    them    joined    together.'
```

This only works with two literals though, not with variables or expressions:

```
prefix ='Py'
prefix    'thon'
```

```
  File "<ipython-input-57-dedab8b7beac>", line 2
    prefix    'thon'
              ^
SyntaxError: invalid syntax
```

SEARCH STACK OVERFLOW

If you want to concatenate variables or a variable and a literal, use +:

```
prefix ='Py'
prefix    +    'thon'
```

```
'Python'
```

```
Strings can be indexed (subscripted), with the first character having index 0.
```

There is no separate character type; a character is simply a string of size one:

```
word='Python'
word[0]         #    character    in    position    0
```

```
'P'
```

```
word[5]         #    character    in    position    5
```

```
'n'
```

```
Indices may also be negative numbers, to start counting from the right.

Note that since -0 is the same as 0, negative indices start from -1.
```

```
word[-1]         #    last    character
```

```
'n'
```

```
word[-2]         #    second-last    character
```

```
'o'
```

```
word[-6]
```

```
'P'
```

```
In addition to indexing, slicing is also supported.

While indexing is used to obtain individual characters, slicing allows
you to obtain substring:
```

```
word[0:2]         #    characters    from    position    0    (included)    to    2    (excluded)
```

```
'Py'
```

```
word[2:5]         #    characters    from    position    2    (included)    to    5    (excluded)
```

```
'tho'
```

```
Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size
of the string being sliced.
```

```
word[:2]              #    character    from    the    beginning    to    position    2    (excluded)
```

```
'Py'
```

```
word[4:]              #    characters    from    position    4    (included)    to    the    end
```

```
'on'
```

```
word[-2:]        #    characters    from    the    second-last    (included)    to    the    end
```

```
'on'
```

Note how the start is always included, and the end always excluded.

This makes sure that s[:i] + s[i:] is always equal to s:

```
word[:2]+word[2:]
```

```
'Python'
```

```
word[:4]+word[4:]
```

```
'Python'
```

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of word[1:3] is 2.

Python strings cannot be changed — they are immutable.

Therefore, assigning to an indexed position in the string results in an error:

```
word[0]    =    'J'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-73-0e57858b2b8a> in <module>
----> 1 word[0]    =    'J'

TypeError: 'str' object does not support item assignment
```

SEARCH STACK OVERFLOW

```
word[2:]='py'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-74-0639537fbf04> in <module>
----> 1 word[2:]='py'

TypeError: 'str' object does not support item assignment
```

SEARCH STACK OVERFLOW

If you need a different string, you should create a new one:

```
'J'+word[1:]
```

```
'Jython'
```

```
word[:2]+'py'
```

```
'Pypy'
```

The built-in function len() returns the length of a string:

```
s='supercalifragilisticexpialidocious'
len(s)
```

```
34
```

Lists

Python knows a number of compound data types, used to group together other values.

The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets.

Lists might contain items of different types, but usually the items all have the same type.

```
squares=[1, 4,9,16,25]
squares
```

```
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in sequence types), lists can be indexed and sliced:

```
squares[0]         #    indexing    returns    the    item
```

```
1
```

```
squares[-1]
```

```
25
```

```
squares[-3:]         #    slicing    returns    a    new    list
```

```
[9, 16, 25]
```

All slice operations return a new list containing the requested elements.

This means that the following slice returns a shallow copy of the list:

```
squares[:]
```

```
[1, 4, 9, 16, 25]
```

Lists also support operations like concatenation:

```
squares+[36,    49,    64,    81,    100]
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
cubes=[1,8,27,65,125]         #    something's    wrong    here
4**3         #    the    cube    of    4    is    64,    not    65!
```

```
64
```

```
cubes[3]=64         #    replace    the    wrong    value
```

```
cubes[3]=64          #    replace    the    wrong    value
cubes
```

```
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the append() method

```
cubes.append(216)         #    add    the    cube    of    6
cubes.append(7    **    3)         #    and    the    cube    of    7
cubes
```

```
[1, 8, 27, 64, 125, 216, 216, 343]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
letters=['a','b','c','d','e','f','g']
letters
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

```
#    replace    some    values
letters[2:5]=['C','D','E']
letters
```

```
['a', 'b', 'C', 'D', 'E', 'f', 'g']
```

```
#    now    remove    them
letters[2:5]=[]
letters
```

```
['a', 'b', 'f', 'g']
```

```
#    clear    the    list    by    replacing    all    the    elements    with    an    empty    list
letters[:]    =    []
letters
```

```
[]
```

The built-in function len() also applies to lists:

```
letters=['a','b','c','d']
len(letters)
```

```
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
a =['a','b','c']
```

```
n=[1,2,3]
```

```
x =[a,n]
x
```

```
[['a', 'b', 'c'], [1, 2, 3]]
```

```
x[0]
```

```
['a', 'b', 'c']
```

```
x[0][1]
```

```
'b'
```

First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together.

For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

```
#    Fibonacci    series: the    sum    of    two    elements    defines    the    next
#The first line contains a multiple assignment : the variables a and b simultaneously get the new values 0 and 1.
#The right-hand side expressions are evaluated from the left to the right.
a,b=0,1
# The body of the loop is indented: indentation is Python’s way of grouping statements.
while(a<10):
  print(a)
  a, b = b, a+b
```

```
0
1
1
2
3
5
8
```

```
#The keyword argument end can be used to avoid the newline after the output, or end the output with a different string:
a,b=0,1
while(a<1000):
  print(a,end=',')
  a, b = b, a+b
```

```
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

✓ 0s completed at 5:55 AM