Introduction to Python

1

Overview of Python

A really cool scripting language

1.1 What is Python

"Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.

Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together."

Further reading:
https://www.python.org/doc/essays/blurb/

Translation

INTERPRETED

Translates code (.py) to Bytecode (.pyc) for the Interpreter, which then compiles the code (.pyc) for the native machine.

OBJECT ORIENTED

A programming paradigm based on Objects, which are data structures that contain data and procedures.

Translation

DYNAMIC SEMANTICS

The dynamic semantics of a language defines what happens when you run a program.

DYNAMIC TYPING

You do not have to specify the variable type (e.g. int, num, long, etc). In Statically Typed languages, you do.

DYNAMIC BINDING

Like Dynamic Typing. Variable type determined at runtime.

Strong vs Dynamic Typing

Python is strongly, dynamically typed.

STRONG TYPING

Strong typing means that the type of a value doesn't suddenly change. A string containing only digits doesn't magically become a number, as may happen in Perl. Every change of type requires an explicit conversion.

DYNAMIC TYPING

Dynamic typing means that runtime objects (values) have a type, as opposed to static typing where variables have a type.

The Python Interpreter

There are four steps that Python Interpreter takes: lexing, parsing, compiling, and interpreting.

1. **Lexing** is breaking the line of code you just typed into tokens.

2. The **parser** takes those tokens and generates a structure that shows their relationship to each other (in this case, an Abstract Syntax Tree).

The Python Interpreter

3. The compiler then takes the Abstract Syntax Tree and turns it into one (or more) code objects.

4. Finally, the interpreter takes each code object executes the code it represents.

Further reading:
http://akaptur.com/blog/2013/11/15/introduction-to-the-python-interpreter/

Is Python a scripting language?

“It's worth noting that languages are not interpreted or compiled, but rather language implementations either interpret or compile code.”

Further reading:
http://programmers.stackexchange.com/questions/46137/what-is-the-main-difference-between-scripting-languages-and-programming-language

Implementations

- CPython – default
- PyPy – JIT Compiler
- Jython – JVM

Further reading:
https://wiki.python.org/moin/PythonImplementations

Everything is an object

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute `__doc__`, which returns the doc string defined in the function's source code.

Further reading:
https://docs.python.org/3.1/reference/datamodel.html

The Zen of Python

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Further reading:
https://www.python.org/dev/peps/pep-0020/

Python has great quotes

“A Foolish Consistency is the Hobgoblin of Little Minds”

Further reading:
https://www.python.org/dev/peps/pep-0008/

1.2 Why use Python

"Python has been an important part of Google since the beginning, and remains so as the system grows and evolves. Today dozens of Google engineers use Python, and we're looking for more people with skills in this language."

Peter Norvig, Google

"Python is fast enough for our site and allows us to produce maintainable features in record times, with a minimum of developers,"

Cuaong Do, Youtube

Pro's

- Easy to read
- Simplistic syntax
- Virtually no boilerplate

Further reading: http://www.infoworld.com/article/2887974/application-development/a-developer-s-guide-to-the-pro-s-and-con-s-of-python.html

Python’s main advantage

COLLABORATION

Because Python’s syntax and style are easy to read, collaborating on a coding project inside a dev team is a easier and more enjoyable process.

You can more quickly intuit another dev’s intent because their code is easier to read.

Hello, World!

Java

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

Python

```python
print ‘Hello, World’
```

"Code is more often read than it is written"

Guido Van Rossum, Python

Further reading:
https://www.python.org/dev/peps/pep-0008/

Con’s

- Weak in mobile computing
- Python isn't in Web browsers
- Consider speed options available

1.3 Python vs other languages

Python vs Java

- Python is easier to read – **collaboration!**
- Java Hotspot is faster than PyPy
- Indentation vs braces
- Java is more portable – JVM

Further reading:
https://blog.udemy.com/python-vs-java/

Python vs Ruby

The difference is mostly cultural.

Ruby devs prize freedom and rapid change, even at the cost of breaking legacy code.

Python devs admire more stable yet slower releases over shiny new features.

Further reading:
https://www.scriptrock.com/articles/python-vs-ruby
http://ruby-doc.org/docs/ruby-doc-bundle/FAQ/FAQ.html

Python 2 vs Python 3

In 2008, Python 3 was released and it was announced the Python 2 would **not** be continued.

- Better unicode support
- Division with integers (e.g. 5/2)

Further reading:
http://blog.teamtreehouse.com/python-2-vs-python-3
http://www.infoworld.com/article/2619428/python/van-rossum--python-is-not-too-slow.html

3

Python – The Basics

“And now, for something different…“

3.1 Basic syntax

PEP 0008

PEP-8 is the style guide for Python Code, authored by Guido Van Rossum (BDFL), Barry Warsaw, and Nick Coghlan.

It is considered best practice to write Python code using the PEP-8 guidelines.

Further reading:
https://www.python.org/dev/peps/pep-0008/

Comments

Comments start with a # character, Unix style. Multiline comments simply all start with #.

```
# This is a comment

# This is a
# multi-line comment

a = 5 * 9                              # 5 times 9
```

Indentation

Indentation is Python's way of grouping statements, as opposed to braces. Each line within a basic block must be indented by the same amount.

```
>>> b = 1
... while b < 10:
...     print b
...     a = b
...     b = a + b
```

Colons

If statements, loops, function definitions, and class definitions all use colons to declare the start of an indented block.

```
>>> b = 1
... while b < 10:
...     print b
...     a = b
...     b = a + b
```

Further reading:
https://docs.python.org/2/faq/design.html#why-are-colons-required-for-the-if-while-def-class-statements

Arithmetic operators

+	Addition	5 + 5 = 10
–	Subtraction	10 - 5 = 5
*	Multiplication	5 * 2 = 10
/	Division	10 / 2 = 5
%	Modulus	10 % 5 = 0
**	Exponent	3 ** 2 = 9

Comparison operators

```
a = 5, b = 6
```

==	is equal to	a == b (false)
!=	not equal to	a != b (true)
>	greater than	a > b (false)
<	less than	a < b (true)
>=	greater than or equal to	a >= b (false)
<=	less than or equal to	a <= b (true)

Python as a calculator

In interactive mode, if you type in a calculation, Python will evaluate it when you press enter

```
>>> 3 * 10 / 2 - 3
12
```

But be careful…

```
>>> 5 / 2
2
```

Wait, what?!

Python as a calculator

Python received an integer from you, and believes that you should receive one back.
If you want to perform floating point arithmetic, at least one of the operands needs to be a float.

```
>>> 5.0 / 2
2.5
```

3.2 Variables

Initializing variables

The equals sign = is used to assign a value to a variable

```
>>> ben = 35
>>> mike = “A really cool guy”
```

TIP: To print out a variable’s value, just type its name and press enter

```
>>> ben
35
```

Initializing variables

A value can be assigned to several variables simultaneously

```
>>> x = y = z = 0
```

Multiple variables can be initialized on the same line, with different values

```
>>> a, b = 0, 1
```

Variable types

Python has four main numeric types:

- plain integers – 32 bit precision
- long integers – unlimited precision
- floating point numbers – double in C
- complex numbers – real & imaginary part

Casting variables

Use the float(), int(), or long() functions to cast a variable to the value of that type

```
>>> a,b,c = 5,10.5,15
>>> float(a)
5.0

>>> int(b)
10.5

>>> long(c)
15L
```

Strings

Assign strings with =, or just type them out

```
>>> "Good afternoon"
'Good afternoon'

>>> 'Good evening'
'Good evening'

>>> "'hello', he said"
"'hello', he said"

>>> 'won\'t you stay for dinner?'
"won't you stay for dinner?"
```

Concatenation & repetition

Strings can be concatenated with + repeated with *

```
>>> a = "Blessed are the "
>>> b = "Cheesemakers"

>>> a + b
'Blessed are the Cheesemakers'

>>> a + b*2
'Blessed are the CheesemakersCheesemakers'
```

Multi-line strings

String literals can span multiple lines in several ways. Continuation lines can be used, with a backslash \ as the last character on the line indicating that the next line is a logical continuation of the line:

```
>>> great_quote = "Blessed are \
... the Cheesemakers

>>> print great_quote
'Blessed are the Cheesemakers'
```

Multi-line strings

Strings can be surrounded in a pair of matching triple-quotes: `"""` or `'''`. End of lines do not need to be escaped when using triple-quotes, but they will be included in the string.

```
>>> great_quote = """Blessed are
... the Cheesemakers"""

>>> print great_quote
Blessed are
the Cheesemakers
```

3.3 If statements

If statements

There can be zero or more elif parts, and using else is optional.

```
>>> x = 5
>>> if x > 4:
...     print "X is greater than four"
... elif x < 4:
...     print "X is less than four"
... else:
...     print "X is something else"

X is greater than four
```

and & or

In Python, we use and and or to perform Boolean operations, not && or ||

```
>>> a = 10
... if a == 5 or a == 10:
...     print 'yes'
yes

>>> a = 10
... if a % 10 == 0 and a % 5 == 0:
...     print 'yes'
yes
```

not

To negate a statement, use not

```
>>> a = 10
... if not a == 10:
...      print ‘a does not equal 10’
... else:
...       print ‘a equals 10’

a equals 10
```

3.4 Loops

The while loop

The while loop will continue looping as long as the condition it's testing for evaluates to true.

```
>>> b = 1
>>> while b < 10:
...     print b
...     a = b
...     b = a + b

1
2
4
8
```

The for loop

The for statement in Python differs a bit from what you may be used to in Java.

Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as Java), Python’s for statement iterates over the **items of any sequence** (a list or a string), in the order that they appear in the sequence.

The for loop

We'll create a sequence for the for loop to iterate over using the range statement.

```
>>> range(5)
[0, 1, 2, 3, 4]

>>> for a in range(5):
...      print a

0
1
2
3
4
```

The for loop

But we could just have easily have used strings

```
>>> greeting = "Hello"

>>> for greet in greeting:
...     print greet

H
e
l
l
o
```

The break statement

The break statement, borrowed from C, breaks out of the smallest enclosing for or while loop.

```
>>> for a in range(5):
...     if a > 2:
...         break
...     print a

0
1
2
```

The continue statement

The continue statement, also borrowed from C, continues with the next iteration of the loop.

```
>>> for a in range(5):
...     if a == 2:
...         continue
...     print a

0
1
3
4
```

3.5 Functions

Defining functions

To define a function:

```
def my_function_name(params):
```

The keyword def introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

Docstrings

The first statement of the function body can optionally be a string literal. This string literal is the function's documentation string, or **docstring**.

```
def return_root_name(params):
    """Returns the root name of dir"""
```

Further reading:
https://www.python.org/dev/peps/pep-0257/

Docstrings

There are tools which use docstrings to automatically produce online or printed documentation (e.g. Sphinx), or to let the user interactively browse through code.

It’s good practice to write docstrings for your functions, so that other programmers can easily understand what your intention was.

Further reading:
http://sphinx-doc.org/

Using parameters

A function may be declared with multiple parameters, which must be used in the function call.

```
def sum_of_params(a, b):
    print a + b

sum_of_params(3, 5)
8
```

Further reading:
https://docs.python.org/2/tutorial/controlflow.html#defining-functions

Default argument values

A function can be declared with default values set for its parameters. This way, the function can be called with fewer arguments than it is defined to allow.

```
def sum_of_params(a, b=5):
     print a + b

sum_of_params(3)
5
```

Further reading:
https://docs.python.org/2/tutorial/controlflow.html#default-argument-values

Calling the function

To call the function, simply use its name, followed by parentheses which are filled with the correct number of parameters.

```
def breakfast(a, b, c, d):
    print a, b, c, “and”, d

breakfast(“spam”, “spam”, “sausage”, “spam”)
spam spam sausage and spam
```

Multiple arguments

Functions can be declared with multiple arguments. We use * to denote a list the the function definition, and ** to denote a dict.

```
>>> def breakfast(a, b, *c):
...     print a, b, c

>>> breakfast('spam', 'sausage', ['eggs', 'bacon'])
spam sausage (['eggs', 'bacon'],)
```

4

Data Structures

"My hovercraft is full of eels!"
Hungarian with translating book at the Tobacconist [spam]

4.1 Lists

Lists

Lists are a data type that are used to group together other values. Lists can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```
>>> list = ['spam', 'eggs', 100, 1234]
>>> list
['spam', 'eggs', 100, 1234]

>>> list[0]
‘spam’
```

List methods

Append x to the end of list

```
list.append(x)
```

Extend the list by appending another list

```
list.extend(L)
```

Insert item x at index i

```
list.insert(i, x)
```

List methods

Remove the first item of value x from list

```
list.remove(x)
```

Sort the items of the list

```
list.sort()
```

Reverse the elements of the list in place

```
list.reverse()
```

List methods

Return the number of times x appears in list

```
list.count(x)
```

Remove the item at the given position in list, and return it. If no index is specified, pop() removes and returns the last item in the list.

```
list.pop(i)
```

Further reading:
https://docs.python.org/2/tutorial/datastructures.html

Using lists as stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”).

To add an item to the top of the stack, use list.append(). To retrieve an item from the top of the stack, use list.pop() without an explicit index.

Using lists as stacks

```
>>> a = [5, 6]
>>> a.append(7)

>>> a
[5, 6, 7]

>>> a.pop()
7

>>> a
[5, 6]
```

Using lists as queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first- out”), however lists are not efficient for this purpose.

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends.

4.2 Tuples

Tuples

Lists and Strings are both of data type "sequence". Tuples are also of type "sequence", although Tuples have different rules to the others.

```
>>> t = (111, 222, ’hello!’)

>>> t
(111, 222, ’hello!’)

>>> t[0]
111
```

Tuples – rules

Tuples may be nested.

```
>>> t = (1, 2, (4, 5))

>>> t
(1, 2, (4, 5))

>>> t[2]
(4, 5)
```

Tuples – rules

Tuples are immutable.

```
>>> t = (“welcome”, 2, 3)

>>> t[0] = “hello”

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: ’tuple’ object does not support
item assignment
```

Tuples – rules

…but tuples can contain mutable objects.

```
>>> t = (1, 2, [3, 4])

>>> t
[1, 2, [3, 4]]

>>> t[2][0] = 5

>>> t
[1, 2, [5, 4]]
```

Tuples – rules

To create an empty tuple, just use empty parens

```
>>> empty = ()
```

To create a tuple with only 1 item, use a comma

```
>>> singleton = (“hello”,)
```

When to use tuples

- Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list.
- Tuples make your code safer by “write-protecting” data that does not need to be changed.

4.3 Sets

Sets

A set is an unordered collection with no duplicate elements.

```
>>> names = ['adam', 'ben', 'charles']

>>> unordered = set(names)

>>> unordered
set(['charles', 'ben', 'adam'])
```

Sets

Sets can be created by using the set() function, or by using curly braces {}.

```
>>> names = ['adam', 'ben', 'charles']
>>> unordered = set(names)

>>> unordered
set(['charles', 'ben', 'adam'])

>>> no_order = {'adam', 'ben', 'charles'}

>>> no_order
set(['charles', 'ben', 'adam'])
```

Set operations

Set objects support mathematical operations like union, intersection, difference, and symmetric difference.

```
>>> s = set('abc')
>>> t = set('bcd')

>>> s
set(['a', 'c', 'b'])

>>> t
set(['c', 'b', 'd'])
```

Set operations

Show items that are in s, but not in t

```
>>> s - t
set(['a'])
```

Show items that are either in s or t

```
>>> s | t
set(['a', 'c', 'b', 'd'])
```

Set operations

Show items that are in both s and t

```
>>> s & t
set(['c', 'b'])
```

Show only items that are NOT in both s and t

```
>>> s ^ t
set(['a', 'd'])
```

4.4 Dictionaries

Dictionaries

Dictionaries are indexed by keys, which can be any immutable type. Strings and numbers can be used as keys.

```
>>> a = dict([('name', 'ben'), ('age', 35)])

>>> a
{'age': 35, 'name': 'ben'}
```

Dictionaries

Dictionaries can be created with curly braces {} or with the dict() function.

```
>>> b = {'age': 20, 'name': 'charles'}
>>> b
{'age': 20, 'name': 'charles'}

>>> a = dict([('name', 'ben'), ('age', 35)])
>>> a
{'age': 35, 'name': 'ben'}
```

Dictionaries – rules

- Dictionaries are called "associative arrays" in other languages
- Keys must be unique
- A pair of braces creates an empty dictionary: `{}`
- It is possible to delete a key:value pair with `del`
- If you store a value using a key that is already in use, the value is overwritten

Dictionary functions

Returns a list of all the keys used in the dictionary

```
>>> a.keys()
['age', 'name']
```

Returns a list of all the values in the dictionary

```
>>> a.values()
[35, 'ben']
```

5

Built-in functions

“My hovercraft is full of eels!”
Hungarian with translating book at the Tobacconist [spam]

5.1 Type

Type

Use the type() built-in to ascertain the type of an object. Whenever in doubt, use type!

```
>>> a = {'name': 'ben', 'age': 35}

>>> a
{'age': 35, 'name': 'ben'}

>>> type(a)
<type 'dict'>
```

5.2 Slicing

Slicing

Slicing returns the items between the specified indices in an object. Slicing can be used for anything that supports numeric indexing. E.g. Strings, Lists, Tuples, etc

```
>>> a = ['tim', 32, 'bob', 27.6, 'mark']

>>> a[2:4]
['bob', 27.6]
```

Slicing

Slicing returns a shallow copy of a list, which means that every slice returns an object which has a new address in memory, but its elements would have the same addresses that elements of source list have.

Slice notation

```
a[i:p]        #items i through p
a[i:]         #items i until end of the object
a[:p]         #items from the start of object through
p
a[:]          #a shallow copy of the whole object

a[i:p:step]  # i through p, using a step
```

5.3 Unicode object

Unicode object

In Python 3, all strings are Unicode by default, but in Python 2, we must explicitly define them.

```
>>> a = unicode('hello there')
>>> a
u'hello there'

>>> a.encode('utf-8')
'hello there'
```

5.4 Range

Range

The range statement generates lists containing arithmetic progressions. It has a finishing point, and optionally a starting point and a step.

```
>>> range(5)
[0, 1, 2, 3, 4]

>>> range(5, 10)
[5, 6, 7, 8, 9]

>>> range(3, 10, 2)
[3, 5, 7, 9]
```

Range

Range can be combined with other functions, like len() to provide useful results.

```
>>> a = ['life', 'is', 'good']

>>> for i in range(len(a)):
...     print i, a[i]

0 life
1 is
2 good
```

5.5 Pass

Pass

The Pass statement creates an empty placeholder that can be used to indicate the the appropriate code hasn’t been written yet.

```
for r in range(10):
    pass

def sell_item(price):
    pass
```

5.6 In, not in

In, not in

The in keyword tests whether a sequence contains a certain value. The not keyword negates in.

```
>>> a
['life', 'is', 'good']

>>> 'life' in a
True

>>> 'taxes' not in a
True
```

5.7 Del

Del

Use del to delete variables, items from a list or dict, or even slices.

```
>>> a
['life', 'is', 'very', 'good']

>>> del a
>>> del a[:2]

>>> b = {'name': 'ben', 'age': 35}
>>> del b['age']
```

6

Advanced looping

6.1 List comprehensions

List comprehensions

List comprehensions provide a concise way to create lists. A common application is to make a new list where each element is the result of some operations applied to each member of another sequence or iterable.

```
>>> squares = [x**2 for x in range(10) if x > 3]

>>> squares
[16, 25, 36, 49, 64, 81]
```

6.2 Enumerate

Enumerate

Returns an enumerate object, which you can use as an index in a loop

```
>>> guests = ['adam', 'ben', 'charles']

>>> for i, guest in enumerate(guests):
...     print i, guest

0 adam
1 ben
2 charles
```

6.3 Loop over multiple lists

Multiple lists

You can loop over multiple sequences with zip()

```
>>> keys = ['name', 'quest', 'favorite color']
>>> values = ['lancelot', 'the holy grail', 'blue']

>>> for key, val in zip(keys, values):
...       print 'My {0} is {1}.'.format(key, val)
```

6.1 Sorted and reverse

Sorted and Reverse

You can loop over a sorted or reversed sequence with sorted() and reversed()

```
>>> attributes = ['name', 'quest', 'favorite color']

>>> for attr in sorted(attributes):
...       print attr

>>> for attr in reversed(attributes):
...       print attr
```

6.1 Iteritems

Iteritems

Return an iterator over the dictionary’s (key, value) pairs.

```
>>> menu = {"starters": "scallops", "mains": "steak",
"dessert": "cake"}

>>> for key, val in menu.iteritems():
...     print key, val

starters scallops
dessert cake
mains steak
```

7 Strings and Regex

7.1 String interpolation

Interpolation

To inject the values of variables into a string, use format(). It is less memory heavy than +.

```
>>> a = "apple"
>>> b = "banana"

>>> "Get me an {}-{} fritter".format(a, b)
"Get me an apple-banana fritter"
```

Interpolation

To inject the values of variables into a string, use format(). It is less memory heavy than +.

```
>>> a = "apple"
>>> b = "banana"

>>> "Get me an {}-{} fritter".format(a, b)
"Get me an apple-banana fritter"
```

Interpolation

The format command can take positional arguments.

```
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs

>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```

Interpolation

It can also identify values in the list with keyword arguments.

```
>>> print 'This {food} is {adjective}.'.format(food='chocolate,
adjective='delicious')

This chocolate is delicious.
```

Interpolation

It can also identify values in the list with keyword arguments.

```
>>> print 'The story of {0} and {other}.'.format('John', other='
Ringo')
The story of John and Ringo.
```

7.2 Str and repr

Str & repr

To convert any value to a string, use the repr() or str() functions.

Str

The str() function is meant to return representations of values which are fairly human-readable, while repr() is meant to generate representations which can be read by the interpreter (or will force a SyntaxError if there is no equivalent syntax).

7.2 StringIO

StringIO

In some cases, you may want to write a string to a memory file (aka. buffer), add to it, and then later read from it.

```
import StringIO

file = StringIO.StringIO()
file.write("Blessed are the cheesemakers ")
file.write("Did he say cheese makers?")

print file.getvalue()          #read the file
file.close()                  #close the file
```

7.3 String operations

String operations

Test if a string contains a substring

```
>>> a = "bunch of coconuts"
>>> a in "I've got a lovely bunch of coconuts"
True
```

Convert to upper case

```
>>> "good morning, sir".upper()
'GOOD MORNING, SIR'
```

Convert to lower case

```
>>> "GOOD MORNING, SIR".lower()
'good morning, sir'
```

String operations

Count the occurrences of a character or sequence

```
>>> 'good morning, sir'.count('o')
3
```

Search a string for the first occurrence of substr

```
>>> 'good morning, sir'.find('morn')
5
```

Convert to lower case

```
>>> "GOOD MORNING, SIR".lower()
'good morning, sir'
```

String operations

Replace all occurrences of substr with another string

```
>>> Good Good Birthday'.replace('Good', 'Happy')
Happy Happy Birthday
```

Cast to string

```
>>> str(9999)
'9999'
```

7.4 Regular Expressions

Regular expressions

We use regular expressions to match patterns inside of Strings.

The re module provides Perl-style regular expression patterns which you can use to match expressions.

```
import re
```

Regex – use raw strings

PROBLEM

Python interprets special characters inside of regular strings. E.g. `\n` means new line. But the re module also uses the `\` character and gives it special meaning, so there can often be a conflict.

Regex – use raw strings

SOLUTION

Raw strings strip away all meaning from a string, so that the compiler doesn't misunderstand your intention. Prefix your string with `r` to turn it into a raw string.

```
>>> r'this is a raw stri\ng'
```

Perl Regex Syntax

^	Beginning of string
$	End of string
.	Any character except new line
*	Match 0 or more times
+	Match 1 or more times
?	Match 0 or 1 times

Perl Regex Syntax

|	Alternative
()	Grouping
[]	Set of characters
{}	Repetition modifier
/	Quote or special character

Perl Regex Syntax

*a**	Zero or more *a*'s
a+	One or more *a*'s
a?	Zero or one *a*'s
a{m}	Exactly m *a*'s
a{m,}	At least m *a*'s
a{m,n}	At least m, but at most n *a*'s

Perl Regex Syntax

\w	any word character
\W	any non-word character
\s	any whitespace character
\S	non non-whitespace character
\d	any digit character
\D	any non-digit character

Regex - match

Match works by finding matches at the beginning of a string.

```
>>> re.match(r'cat', 'cats dogs')
<_sre.SRE_Match object at 0x10ab9cb28>

>>> re.match(r'cat', 'dogs cats')
```

Regex – valid syntax

Any valid Perl-based regex syntax can be used.

```
person = "John Doe. 35."

result = re.match(r'\w+ \w+. \d+.',
person)

result.group(0)
```

Regex - group

When called with 0 as its argument, group will return the pattern matched by the query

```
result = re.match(r'cat', 'cats dogs')

result.group(0)
'cat'
```

Regex – search

Search is not restricted to the beginning of a string, so it will find a match anywhere, but it will stop searching after it has found the first occurrence of the pattern.

```
result = re.search(r'cat', 'dogs cats')

result.group(0)
'cat'
```

Regex – findall

Finds all occurrences of the pattern in a string, and returns a list matching all patterns.

```
>>> re.findall(r'spam', 'spam eggs spam')
['spam', 'spam']
```

Regex - match objects

Match and Search return a match object. This object contains information about the result, such as the starting and finished indices of the found pattern.

```
>>> match = re.search(r'dog', 'dog cat dog')

>>> match.start()
0

>>> match.end()
3
```

Regex – grouping

Grouping enables us to target certain parts of the regex match, and then work with them individually.

```
person = "John Doe. 35."

result = re.match(r'(\w+) (\w+). (\d+).',
person)

result.group(0)
```

Regex – named groups

Python also lets us name the groups that we define, for ease of use with larger strings.

```
person = "John Doe. 35."

result = re.match(r'(?P<first>\w+) (?
P<last>\w+). (?P<age>\d+).', person)

result.group('age')
'35'
```

Regex – compile

A cleaner way to work is to compile your pattern into a regular expression object.

This way, it can be reused easily, and other functions become available to it.

```
regex_obj = re.compile(r'\w+')

regex_obj
<_sre.SRE_Pattern object at 0x108c41ae0>
```

Regex – search & replace

Use sub() to replace every occurrence of a pattern inside a string.

```
>>> a = 'john doe'
>>> obj = re.compile(r'jo\w+')

>>> obj.sub('dave', a)
'dave doe'
```

8

Exception Handling

“My hovercraft is full of eels!”
Hungarian with translating book at the Tobacconist [spam]

8.1 Error types

Error types

There are (at least) two distinguishable kinds of errors: syntax errors and exceptions. Syntax errors occur during the parsing stage, while Exceptions occur later during the compilation phase.

Syntax errors

When your code is syntactically incorrect and you run it, the parser repeats the offending line, and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected. File name and line number are printed for convenience.

```
>>> a = 'hello"
  File "<stdin>", line 1
    a = 'hello"
              ^
SyntaxError: EOL while scanning string
literal
```

8.2 Exceptions

Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal, because you, and your program, can handle them.

The try statement

1. First, the try clause (the statement(s) between the try and except keywords) is executed.

2. If no exception occurs, the except clause is skipped and execution of the try statement is finished.

The try statement

3. If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.

The try statement

4. If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.

The try statement

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed.

Error handling demo

```
>>> try:
...     10 / 0
... except ZeroDivisionError as err:
...     print "ZeroDivisionError"

ZeroDivisionError
```

Else

The try...except statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception.

```
try:
    10 / 0
except ZeroDivisionError as err:
    print "ZeroDivisionError"
else:
    print "No errors were encountered"
```

Finally

The try statement has another optional clause, finally, which is intended to define clean-up actions that must be executed under all circumstances.

```
try:
    raise KeyboardInterrupt
finally:
    print 'thats all folks'

thats all folks
```

Raising exceptions

The raise statement allows the programmer to force a specified exception to occur.

```
>>> raise NameError("Well that’s a silly
name")

Traceback (most recent call last):
    File "<stdin>", line 1, in ?
NameError: Well that’s a silly name
```

Exceptions arguments

Arguments may be passed to an Exception, and then read from the Exception during handling.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print type(inst)
...     print inst.args
...     print inst
...     x, y = inst.args
...     print 'x =', x
...     print 'y =', y

<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

User defined exceptions

A user may define their own exceptions by deriving from the Exception class

```
class UserError(Exception):
     def __init__(self, value):
          self.value = value
     def __str__(self):
          return repr(self.value)

raise UserError('Oops')
```

User defined exceptions

Output

```
try:
    raise UserError(2*2)
except UserError as e:
    print 'My exception occurred, value:', e.value

My exception occurred, value: 4
```

Exception naming

Most exceptions are defined with names that end in “Error,” similar to the naming of the standard exceptions.

Common Exceptions

ZeroDivisionError

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo
by zero
```

Common Exceptions

NameError

```
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
```

Common Exceptions

TypeError

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int'
objects
```