Sheet of Python

Welcoming all programmers

Python Training

Sessions by Tiago Montes

Running Python

command	result
`python`	Brings up the Python REPL. Exit with EOF, `exit()` or `quit()`.
`python` *filename.py*	Runs the Python program in *filename.py*.
`python -i` *filename.py*	Runs the Python program in *filename.py* and, when done, enters the REPL where the program state can be inspected.

Exploring

expression	notes
`type(`*obj*`)`	Returns the type of *obj*.
`dir(`*obj*`)`	Returns a list with the names of the attributes of *obj*.
obj`.`*attr*	The value of the attribute *attr* on object *obj*.
`help(`*obj*`)`	Displays interactive help on *obj*. Hit **q** to get back to the REPL.
`help()`	Enters interactive help, with a **help>** prompt. (example, type **modules** to get a list of importable modules)

Everything is an object. Objects have a type and attributes.

Assignment

statement	notes
`name =` *expression*	**name** becomes reference to result of *expression*; no data is copied.
`del name`	Deletes the reference.
`a, b =` *expression*	Iterates over the result of *expression*: assigns first value to **a**, second to **b**. Fails if iteration does not produce two values.
`a, *b =` *expression*	Iterates over the result of *expression*: assigns first value to **a**, remaining zero/more values put in a new list, assigned to **b**.

At play in imports, function/class definitions and calls, in for loops, comprehensions, and more.

Working with Files

statement	returns
`open(`*fn*`,` *mode*`)`	Open file, named *fn* in *mode* `'r'`ead, `'w'`rite, `'t'`ext, `'b'`inary. Use the optional **encoding** argument when in text mode.
f`.read(`*size*`)`	Up to *size* long **str**/**bytes** read from the file. Empty on EOF.
f`.write(`*payload*`)`	Count of written **str**/**bytes**, from *payload*, to the file.

Avoid the **close** method and use the **with** statement with file objects, instead.

In the Standard Library: **pathlib**, **os**/**os.path**, **shutil**, **zipfile**, **gzip**, **bz2**, **csv**, and more.

Builtin Types

name	false / empty	examples of other values	some useful methods	notes
`bool`	`False`	`True`	-	Behaves like **int** `0` and `1`, respectively.

`int`	`0`	`-42`
		`7_654_321`
		`0xBECA`
			`bit_length`, `to_bytes`, `from_bytes`	Unlimited precision.
				Digit grouping with _ is Python 3.6 or later.

`float`	`0.0`	`4.2`
		`6.626e-34`
		`float('inf')`
			`is_integer`, `as_integer_ratio`	IEEE-754 floating points, supported by the underlying hardware.

`str`	`''`	`'hello there'`
		`"it's a nice day"`
		`"""there's a "quote" here"""`
			`count`, `encode`, `find`, `format`, `index`, `join`, `lower`, `partition`, `replace`, `split`, `startswith`, `strip`, `upper`	Interpolates `\n` as newline, `\t` as tab, `\N{name}` as a unicode code point, etc.
				`r`-prefixed strings do not interpolate.

`tuple`	`( )`	`('single element',)` `(42, 'name', True)`	`count`, `index`	Tuples are immutable. Use a trailing comma to create single element tuples.
`list`	`[ ]`	`[1, 2, 3]` `['hello', False, 42]`	`append`, `clear`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse`, `sort`	Mutable, contiguous sequences of items.
`dict`	`{ }`	`{'id': 42, 'name': 'jane'}` `{0: 0, 1: 1, 2: 0, 3: 0}`	`clear`, `get`, `items`, `keys`, `pop`, `popitem`, `update`, `values`	Maps keys, which must be hashable/ immutable, to values of any type.
`set`	`set()`	`{1, 2, 3}` `{'h', 'e', 'l', 'o'}`	`add`, `clear`, `discard`, `intersection`, `issubset`, `pop`, `remove`, `union`, `update`	Members must be hashable/immutable.
`NoneType`	`None`	The type only has the **None** value.	-	Used often to represent missing or undefined information.

Immutables	`bool`, `int`, `float`, `str`, `bytes`, and `tuple`; `bytes` are `str`-like, representing numbers from 0 to 255, used in low-level I/O operations.
Sequences	`str`, `bytes`, `list`, and `tuple` have length, are indexable by position via `[`*from_start*`]` / `[-`*from_end*`]`, are sliceable via `[`*start*`:`*stop*`]`, are iterable.
Containers	`tuple`, `list`, `dict`, and `set` hold references to objects, have length, are iterable, support the `in` containment operator.

In the Standard Library: **date**, **time**, and **datetime** in the **datetime** module, **namedtuple**, **defaultdict**, and **deque**, in the **collections** module, and more.

Builtin Functions

name	positional args	some keyword args	returns	usage
`print`	zero or more	**sep** and **end**	**None**	`print('Python', 3, end='!\n')`
`input`	*prompt*	-	**str** with user supplied input, with no trailing `\n`.	`name = input('Your name? ')`
`len`	*obj*	-	**int** with the length of *obj*.	`len('hello') == 5`
`range`	*stop* or *start*, *stop*[, *step*]	-	Iterable range of **int**, from *start* to *stop*-1, in steps of *step*.	`list(range(3)) == [0, 1, 2]`
`sorted`	*iterable*	**key** and **reverse**	New **list** with items in *iterable*, sorted by the result of applying the **key** function to each item.	
`enumerate`	*iterable*	**start**	Iterable of (p, i) where *p* is position and *i* is each item in *iterable*.	Tracks item "position" in an iterable.
`zip`	one or more iterables	-	Iterable of $(i_1, i_2, \ldots)$ with each i_n is obtained from each arg.	Iterates over multiple iterables in parallel.

More, including **any**, **all**, **min**, **max**, **sum**, etc. Types are classes and feel like functions: calling them returns an object of the type, based on the passed in arguments.

In the Standard Library: the **operator** module has useful **key** functions for **sorted**; the **itertools** module contains powerful utilities to work with iterables.

Sheet of Python

Welcoming all programmers

Python Training

Sessions by Tiago Montes

Creating Functions

code	notes

`def func(a, b):`
`  '''doc string'''`
`  ...`
	Creates function named **`func`**, taking two arguments. **`help(func)`** displays docstring. If the function body does not explicitly **`return`**, calling **`func`** returns **`None`**.

`def func(a, b=None):` `  ...`	**`b`** is an optional argument, assigned to **`None`** if omitted in the call. Avoid mutable default argument values.
`def func(*args):` `  ...`	**`func`** accepts arbitrary number of positional arguments; **`args`** is a tuple of the passed in arguments.
`def func(**kwargs):` `  ...`	**`func`** accepts arbitrary number of keyword arguments; **`kwargs`** is a dict of the passed in name/value pairs.

Call parameters can be passed by position and/or by name. Pass ***`*`*iterable** to expand it into multiple by-position arguments, pass ***`**`*mapping** to expand it into multiple keyword/ by-name arguments.
Functions are objects too: can be referenced and passed around like any other object.

Generators

code	notes

`def gen(...):`
`  ...`
`  yield` *expression*
`  ...`
	Function definitions that use one or more **`yield`** statements become generator functions. Calling them returns generator objects.

Generator objects are iterable, and behave like iterators: once iterated, they're done.

Comprehensions

expression	equivalent code

`# list comprehension`
`[` *expr* `for` *target* `in` *iterable* `if` *cond* `]`
	`result = []`
	`for` *target* `in` *iterable*`:`
	`  if` *cond*`:`
	`  result.append(`*expr*`)`

`# dict comprehension`
`{` *ke*`:` *ve* `for` *target* `in` *iterable* `if` *cond* `}`
	`result = {}`
	`for` *target* `in` *iterable*`:`
	`  if` *cond*`:`
	`  result[`*ke*`] =` *ve*

Comprehensions create objects from iterables. The **`if`** clause is optional: when omitted something like **`if True`** is assumed, and no filtering takes place.
Set comprehensions create sets: list comprehension syntax, using **`{ }`** instead of **`[ ]`**.
Generator expressions create generator objects: list comprehensions syntax, using **`( )`** instead of **`[ ]`**. Refer to the **Generators** box, above.

Control Flow and Loops

code	notes

`if` *expression*`:`
`  ...`
`elif` *expression*`:`
`  ...`
`else:`
`  ...`
	Expressions are evaluated in a boolean context, but do not need to evaluate to a **`bool`**: refer to **false / empty** column, in the **Builtin Types** box, on the other page. Use zero, one, or more **`elif`** clauses. The **`else`** clause is optional.

`while` *expression*`:`
`  ...`
`else:`
`  ...`
	Executes repeatedly, as long as ***expression*** evaluates to true, in a boolean context. The **`continue`** statement jumps back to the top, leading to ***expression*** evaluation; **`break`** gets out of the loop. The optional **`else`** block runs if the loop terminates without a **`break`** statement.

`for` *target* `in` *iterable*`:`
`  ...`
`else:`
`  ...`
	Iterates over ***iterable***, obtaining one ***value*** at a time. The loop block is run once for each such ***value***, assigned to ***target***. The loop ends when the iteration ends. The **`continue`**/**`break`** statements and the optional **`else`** block work the same way they do in **`while`** loops.

Exception Handling

code	notes

`try:`
`  ...`
`except` *etype* `as` *eobj*`:`
`  ...`
`else:`
`  ...`
`finally:`
`  ...`
	Runs the **`try`** block. If an exception of type ***etype*** is raised, runs the associated **`except`** block, where ***eobj*** is a reference to the exception object itself (the **`as`** ***eobj*** clause is optional). If no exception is raised in the **`try`** block, the **`else`** block is run. The **`finally`** block runs last, no matter what. Multiple **`except`** blocks can be used. The **`else`** block is optional. The **`except`** and **`finally`** blocks are optional, but one must be present.

Exceptions raised outside the **`try`** block are not handled. A single **`except`** clause can handle different exceptions when ***etype*** is a tuple of different exception types.
Uncaught exceptions generate stack trace outputs towards the standard error, where the last line that indicates the exception type and message.
Exceptions can be explicitly raised with the **`raise`** statement.

Iteration

code	notes
`iter(`*obj*`)`	Returns an iterator over ***obj***. Fails if ***obj*** is not iterable.
`next(`*iterator*`)`	Returns the next iteration value tracked by ***iterator***. Raises **`StopIteration`** when the iterator is done.

Manual iteration not often needed: **`for`**/comprehensions are enough. Iterators are created to iterate over iterables: they just go forward and can't be reused once done.

Iteration works like this all around: iterator obtained first, next called on it until done.

Creating Classes

code	notes
`class C:` `  '''doc string'''`	Creates a class named **`C`**.
`  attr_name = ...`	Assignments become class attributes.
`  def __init__(self, ...):` `  ...`	Initializer: passed the newly created instance via **`self`**. Normally creates attributes in **`self`**.
`  def method(self, ...):` `  ...`	Functions become instance methods: **`self`**, a reference to the instance, auto passed in.

`  @property`
`  def name(self):`
`  ...`
	Getting the ***`.name`*** attribute from an instance, runs this: it's value is whatever is returned.

`  @name.setter`
`  def name(self, value):`
`  ...`
	Setting the ***`.name`*** attribute in an instance runs this, passing in the assigned ***value***. Needs getter, as above: if missing, ***`.name`*** is read-only.

`  def __repr__(self):` `  ...`	Called by Python to obtain a string representation of the instance; see also **`__str__`**.
`  def __eq__(self, other):` `  ...`	Return whether the instance is equal to ***other***. Called by Python on the **`==`** operator.
`  def __add__(self, other):` `  ...`	Called by Python to add the instance to ***other***, when the **`+`** operator is used.
`  def __del__(self):` `  ...`	Called by Python on finalization. Don't use as destructor. Use a context manager instead.

Class instances are created by calling the class as if it was a plain function. Arguments to such calls are passed to the **`__init__`** method, after the auto passed in **`self`**.
Function argument capabilities (default values, arbitrary arguments, etc.) usable here.
Static/class methods created with the **`@staticmethod`**/**`@classmethod`** decorators.
`__named__` methods are called by Python: refer to the Python data model document.
Inherit from other classes with a **`class`** ***name*`(`*base1*`, `*base2*`, ...)`** declaration. Use the **`super`** builtin to access base classes and their attributes and methods.

Modules and Packages

code	notes
`import` *mod*	Finds ***mod.py*** in **`sys.path`** and runs it. When done, ***mod*** is an object where globals in the imported file (functions, classes, variables, ...) become its attributes.
`from` *mod* **`import`** *name*	The same as **`import`** ***mod***, followed by the equivalent of ***name*** = ***mod.name*** and **`del`** ***mod***.

Imports are cached: re-importing does not bring in code updates. Imports not only look for ***mod.py*** files, but also for ***mod/__init__.py***, native shared libraries, and others.