appendix
Python primer for R users

You may find yourself wanting to read and understand some Python, or even port some Python to R. This guide is designed to enable you to do these tasks as quickly as possible. As you'll see, R and Python are similar enough that this is possible without necessarily learning all of Python. We start with the basics of container types and work up to the mechanics of classes, dunders, the iterator protocol, the context protocol, and more!

A.1 Whitespace

Whitespace matters in Python. In R, expressions are grouped into a code block with `{}`. In Python, that is done by making the expressions share an indentation level. For example, an expression with an R code block might be:

```
if (TRUE) {
  cat("This is one expression. \n")
  cat("This is another expression. \n")
}
```

The equivalent in Python:

```
if True:
  print("This is one expression.")
  print("This is another expression.")
```

Python accepts tabs or spaces as the indentation spacer, but the rules get tricky when they're mixed. Most style guides suggest (and IDEs default to) using spaces only.

A.2 Container types

In R, the `list()` is a container you can use to organize R objects. R's `list()` is feature-packed, and there is no single direct equivalent in Python that supports all the same features. Instead there are (at least) four different Python container types you need to be aware of: lists, dictionaries, tuples, and sets.

A.2.1 Lists

Python lists are typically created using bare brackets: `[]`. (The Python built-in `list()` function is more of a coercion function, closer in spirit to R's `as.list()`). The most important thing to know about Python lists is that they are modified in place. Note in the example below that `y` reflects the changes made to `x`, because the underlying list object that both symbols point to is modified in place:

```
x = [1, 2, 3]
y = x
x.append(4)
print("x is", x)
```

y and x now refer to the same list!

```
x is [1, 2, 3, 4]
```

```
print("y is", y)
```

```
y is [1, 2, 3, 4]
```

One Python idiom that might be concerning to R users is that of growing lists through the `append()` method. Growing lists in R is typically slow and best avoided. But because Python's list are modified in place (and a full copy of the list is avoided when appending items), it is efficient to grow Python lists in place.

Some syntactic sugar around Python lists you might encounter is the usage of `+` and `*`. These are concatenation and replication operators, akin to R's `c()` and `rep()`:

```
x = [1]
x
```

```
[1]
```

```
x + x
```

```
[1, 1]
```

```
x * 3
```

```
[1, 1, 1]
```

You can index into lists with integers using trailing `[]`, but note that indexing is 0-based:

```
x = [1, 2, 3]
x[0]
```

```
1
```

```
x[1]
```

```
2
```

```
x[2]
```

```
3
```

```
try:
  x[3]
except Exception as e:
  print(e)
```

```
list index out of range
```

When indexing, negative numbers count from the end of the container:

```
x = [1, 2, 3]
x[-1]
```

```
3
```

```
x[-2]
```

```
2
```

```
x[-3]
```

```
1
```

You can slice ranges of lists using a colon (`:`) inside brackets. Note that the slice syntax is *not* inclusive of the end of the slice range. You can optionally also specify a stride:

```
x = [1, 2, 3, 4, 5, 6]

x[0:2]
```

Get items at index positions 0 and 1, not 2.

```
[1, 2]
```

```
x[1:]
```

Get items from index position 1 to the end.

```
[2, 3, 4, 5, 6]
```

```
x[:-2]
```

Get items from the beginning up to the second to last.

```
[1, 2, 3, 4]
```

```
x[:]                    # Get all the items (the idiom used to copy the list so as not to modify in place).
```

```
[1, 2, 3, 4, 5, 6]
```

```
x[::2]                  # Get all the items, with a stride of 2.
```

```
[1, 3, 5]
```

```
x[1::2]                 # Get all the items from index 1 to the end, with a stride of 2.
```

```
[2, 4, 6]
```

A.2.2 Tuples

Tuples behave like lists, except they are not mutable, and they don't have the same modify-in-place methods like `append()`. They are typically constructed using bare `()`, but parentheses are not strictly required, and you may see an implicit tuple being defined just from a comma-separated series of expressions. Because parentheses can also be used to specify order of operations in expressions like `(x + 3) * 4`, a special syntax is required to define tuples of length 1: a trailing comma. Tuples are most commonly encountered in functions that take a variable number of arguments:

```
x = (1, 2)              # A tuple of length 2
type(x)
```

```
<class 'tuple'>
```

```
len(x)
```

```
2
```

```
x
```

```
(1, 2)
```

```
x = (1,)                # A tuple of length 1
type(x)
```

```
<class 'tuple'>
```

```
len(x)
```

```
1
```

```
x
```

```
(1,)
```

```
x = ()
print(f"{type(x) = }; {len(x) = }; {x = }")
```

A tuple of length 0

Example of an interpolated string literals. You can do string interpolation in R using glue::glue().

```
type(x) = <class 'tuple'>; len(x) = 0; x = ()
```

```
x = 1, 2
type(x)
```

Also a tuple

```
<class 'tuple'>
```

```
len(x)
```

```
2
```

Beware a single trailing comma! This is a tuple!

```
x = 1,
type(x)
```

```
<class 'tuple'>
```

```
len(x)
```

```
1
```

PACKING AND UNPACKING

Tuples are the container that powers the *packing* and *unpacking* semantics in Python. Python provides the convenience of allowing you to assign multiple symbols in one expression. This is called *unpacking*.

For example:

```
x = (1, 2, 3)
a, b, c = x
a
```

```
1
```

```
b
```

```
2
```

```
c
```

```
3
```

You can access similar unpacking behavior from R using `` zeallot::`%<-%` ``.

Tuple unpacking can occur in a variety of contexts, such as iteration:

```
xx = (("a", 1),
      ("b", 2))
for x1, x2 in xx:
  print("x1 =", x1)
  print("x2 =", x2)
```

```
x1 = a
x2 = 1
x1 = b
x2 = 2
```

If you attempt to unpack a container to the wrong number of symbols, Python raises an error:

```
x = (1, 2, 3)
a, b, c = x      <- Success
a, b = x         <- Error: x has too many values to unpack.
```

```
Error in py_call_impl(callable, dots$args, dots$keywords):
➥ ValueError: too many values to unpack (expected 2)
```

```
a, b, c, d = x   <- Error: x has not enough values to unpack.
```

```
Error in py_call_impl(callable, dots$args, dots$keywords):
➥ ValueError: not enough values to unpack (expected 4, got 3)
```

It is possible to unpack a variable number of arguments, using `*` as a prefix to a symbol (We'll see the `*` prefix again when we talk about functions.):

```
x = (1, 2, 3)
a, *the_rest = x
a
```

```
1
```

```
the_rest
```

```
[2, 3]
```

You can also unpack nested structures:

```
x = ((1, 2), (3, 4))
(a, b), (c, d) = x
```

A.2.3 Dictionaries

Dictionaries are most similar to R environments. They are a container where you can retrieve items by name, though in Python the name (called a *key* in Python's parlance) does not need to be a string like in R. It can be any Python object with a `hash()` method (meaning, it can be almost any Python object). They can be created using syntax like `{key: value}`. Like Python lists, they are modified in place. Note that `reticulate::r_to_py()` converts R named lists to dictionaries:

```
d = {"key1": 1,
     "key2": 2}
d2 = d
```

```
d
```

```
{'key1': 1, 'key2': 2}
```

```
d["key1"]
```

```
1
```

```
d["key3"] = 3
d2                     <- Modified in place!
```

```
{'key1': 1, 'key2': 2, 'key3': 3}
```

Like R environments (and unlike R's named lists), you cannot index into a dictionary with an integer to get an item at a specific index position. Dictionaries are *unordered* containers (however, beginning with Python 3.7, dictionaries do preserve the item insertion order):

```
d = {"key1": 1, "key2": 2}
d[1]                        <- Error: The integer "1" is not one of the keys in the dictionary.
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): KeyError: 1
```

A container that closest matches the semantics of R's named list is the `OrderedDict` (http://mng.bz/7y5m), but that's relatively uncommon in Python code, so we don't cover it further.

A.2.4 Sets

Sets are a container that can be used to efficiently track unique items or deduplicate lists. They are constructed using `{val1, val2}` (like a dictionary, but without `:`). Think of them as a dictionary where you use only the keys. Sets have many efficient methods for membership operations, like `intersection()`, `issubset()`, `union()`, and so on:

```
s = {1, 2, 3}
type(s)
```

```
<class 'set'>
```

```
s
```

```
{1, 2, 3}
```

```
s.add(1)
s
```

```
{1, 2, 3}
```

A.3 Iteration with for

The `for` statement in Python can be used to iterate over any kind of container:

```
for x in [1, 2, 3]:
  print(x)
```

```
1
2
3
```

R has a relatively limited set of objects that can be passed to `for`. Python, by comparison, provides an iterator protocol interface, which means that authors can define custom objects, with custom behavior that is invoked by `for`. (We'll have an example for how to define a custom iterable when we get to classes.) You may want to use a Python iterable from R using reticulate, so it's helpful to peel back the syntactic sugar a little to show what the `for` statement is doing in Python, and how you can step through it manually.

Two things happen: first, an iterator is constructed from the supplied object. Then, the new iterator object is repeatedly called with `next()` until it is exhausted:

```
l = [1, 2, 3]
it = iter(l)    <-- Create an iterator object.
it
```

```
<list_iterator object at 0x7f5e30fbd190>
```

Call `next()` on the iterator until it is exhausted:

```
next(it)
```

```
1
```

```
next(it)
```

```
2
```

```
next(it)
```

```
3
```

```
next(it)
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): StopIteration
```

In R, you can use reticulate to step through an iterator the same way:

```
library(reticulate)
l <- r_to_py(list(1, 2, 3))
it <- as_iterator(l)
```

```
iter_next(it)
```

```
1.0
```

```
iter_next(it)
```

```
2.0
```

```
iter_next(it)
```

```
3.0
```

```
iter_next(it, completed = "StopIteration")
```

```
[1] "StopIteration"
```

Iterating over dictionaries first requires understanding whether you are iterating over the keys, values, or both. Dictionaries have methods that allow you to specify which:

```
d = {"key1": 1, "key2": 2}
for key in d:
  print(key)
```

```
key1
key2
```

```
for value in d.values():
  print(value)
```

```
1
2
```

```
for key, value in d.items():
  print(key, ":", value)
```

```
key1 : 1
key2 : 2
```

A.3.1 Comprehensions

Comprehensions are special syntax that allow you to construct a container like a list or a dict, while also executing a small operation or single expression on each element. You can think of it as special syntax for R's `lapply`. For example:

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

l = [element + 100 for element in x]
l
```

A list comprehension built from x, where you add 100 to each element

```
[101, 102, 103]
```

```
d = {str(element) : element + 100
     for element in x}
d
```

A dict comprehension built from x, where the key is a string. Python's str() is like R's as.character().

```
{'1': 101, '2': 102, '3': 103}
```

A.4 Defining functions with def

Python functions are defined with the `def` statement. The syntax for specifying function arguments and default argument values is very similar to R:

```
def my_function(name = "World"):
  print("Hello", name)

my_function()
```

```
Hello World
```

```
my_function("Friend")
```

```
Hello Friend
```

The equivalent R snippet would be:

```
my_function <- function(name = "World") {
  cat("Hello", name, "\n")
}

my_function()
```

```
Hello World
```

```
my_function("Friend")
```

```
Hello Friend
```

Unlike R functions, the last value in a function is not automatically returned. Python requires an explicit return statement:

```
def fn():
  1
print(fn())
```

```
None
```

```
def fn():
  return 1
print(fn())
```

```
1
```

> **NOTE** For advanced R users, Python has no equivalent of R's argument "promises." Function argument default values are evaluated once, when the function is constructed. This can be surprising if you define a Python function with a mutable object as a default argument value, like a Python list!

```
def my_func(x = []):
  x.append("was called")
  print(x)

my_func()
my_func()
my_func()
```

```
['was called']
['was called', 'was called']
['was called', 'was called', 'was called']
```

You can also define Python functions that take a variable number of arguments, similar to ... in R. A notable difference is that R's ... makes no distinction between named and unnamed arguments, but Python does. In Python, prefixing a single `*` captures unnamed arguments, and two `**` signifies that *keyword* arguments are captured:

```
def my_func(*args, **kwargs):
  print("args =", args)          <- args is a tuple.
  print("kwargs =", kwargs)      <- kwargs is a dictionary.

my_func(1, 2, 3, a = 4, b = 5, c = 6)
```

```
args = (1, 2, 3)
kwargs = {'a': 4, 'b': 5, 'c': 6}
```

Whereas the `*` and `**` in a function definition signature *pack* arguments, in a function call, they *unpack* arguments. Unpacking arguments in a function call is equivalent to using `do.call()` in R:

```
def my_func(a, b, c):
  print(a, b, c)

args = (1, 2, 3)
my_func(*args)
```

```
1 2 3
```

```
kwargs = {"a": 1, "b": 2, "c": 3}
my_func(**kwargs)
```

```
1 2 3
```

A.5 Defining classes with class

One could argue that in R, the preeminent unit of composition for code is the `function`, and in Python, it's the `class`. You can be a very productive R user and never use R6, reference classes, or similar R equivalents to the object-oriented style of Python `class`es.

In Python, however, understanding the basics of how `class` objects work is requisite knowledge, because `class`es are how you organize and find methods in Python (in contrast to R's approach, where methods are found by dispatching from a generic). Fortunately, the basics of `class`es are accessible.

Don't be intimidated if this is your first exposure to object-oriented programming. We'll start by building up a simple Python class for demonstration purposes:

```
class MyClass:
  pass      <-- pass means do nothing.

MyClass
```

```
<class '__main__.MyClass'>
```

```
type(MyClass)
```

```
<class 'type'>
```

```
instance = MyClass()
instance
```

```
<__main__.MyClass object at 0x7f5e30fc7790>
```

```
type(instance)
```

```
<class '__main__.MyClass'>
```

Like the `def` statement, the `class` statement binds a new callable symbol, `MyClass`. First note the strong naming convention: classes are typically `CamelCase`, and functions are typically `snake_case`. After defining `MyClass`, you can interact with it, and see that it has type `'type'`. Calling `MyClass()` creates a new object *instance* of the class, which has type `'MyClass'` (ignore the `__main__.` prefix for now). The instance prints with its memory address, which is a strong hint that it's common to be managing many instances of a class, and that the instance is mutable (modified-in-place by default).

In the first example, we defined an empty `class`, but when we inspect it we see that it already comes with a bunch of attributes (`dir()` in Python is equivalent to `names()` in R):

```
dir(MyClass)
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__']
```

A.5.1 What are all the underscores?

Python typically indicates that something is special by wrapping the name in double underscores, and a special double-underscore-wrapped token is commonly called a *dunder*. "Special" is not a technical term; it just means that the token invokes a Python language feature. Some dunder tokens are merely ways that code authors can plug into specific syntactic sugars; others are values provided by the interpreter that would be otherwise hard to acquire; yet others are for extending language interfaces (e.g., the iteration protocol); and, finally, a small handful of dunders are truly complicated to understand. Fortunately, as an R user looking to use some Python features through reticulate, you only need to know about a few easy-to-understand dunders.

The most common dunder method you'll encounter when reading Python code is `__init__()`. This is a function that is called when the class constructor is called, that is, when a class is *instantiated*. It is meant to initialize the new class instance. (In very sophisticated code bases, you may also encounter classes where `__new__()` is also defined; this is called before `__init__()`.)

```
class MyClass:

  print("MyClass's definition body is being evaluated")

  def __init__(self):
    print(self, "is initializing")
```

Note that this is evaluated once, when the class is first defined.

```
MyClass's definition body is being evaluated
```

```
instance = MyClass()
```

```
<__main__.MyClass object at 0x7f5e30fcafd0> is initializing
```

```
print(instance)
```

```
<__main__.MyClass object at 0x7f5e30fcafd0>
```

Note the identical memory address between `instance` and what `self` was in the __init__() method.

```
instance2 = MyClass()
```

```
<__main__.MyClass object at 0x7f5e30fc7790> is initializing
```

```
print(instance2)
```

```
<__main__.MyClass object at 0x7f5e30fc7790>
```

New instance, new memory address

A few things to note:

- The `class` statement takes a code block that is defined by a common indentation level. The code block has the same exact semantics as any other expression that takes a code block, like `if` and `def`. The body of the class is evaluated only *once*—when the class constructor is first being created. Beware that any objects defined here are shared by all instances of the class!
- `__init__()` is just a normal function, defined with `def` like any other function, except it's inside the class body.
- `__init__()` takes an argument: `self`. `self` is the class instance being initialized (note the identical memory address between `self` and `instance`). Also note that we didn't provide `self` when calling `MyClass()` to create the class instance; `self` was spliced into the function call by the language.
- `__init__()` is called each time a new instance is created.

Functions defined inside a `class` code block are called *methods*, and the important thing to know about methods is that each time they are called from a class instance, the instance is spliced into the function call as the first argument. This applies to all functions defined in a class, including dunders. The sole exception is if the function is decorated with something like `@classmethod` or `@staticmethod`:

```
class MyClass:
  def a_method(self):
    print("MyClass.a_method() was called with", self)

instance = MyClass()
instance.a_method()
```

```
MyClass.a_method() was called with <__main__.MyClass object at 0x7f5e30fcadf0>:
```

```
MyClass.a_method()     <-- Error: missing required argument self
```

```
Error in py_call_impl(callable, dots$args, dots$keywords):
➥ TypeError: a_method() missing 1 required positional argument: 'self'
```

```
MyClass.a_method(instance)     <-- Identical to instance.a_method()
```

```
MyClass.a_method() was called with <__main__.MyClass object at 0x7f5e30fcadf0>
```

Other dunders worth knowing about are:

- `__getitem__`—The function invoked when extracting a slice with `[` (equivalent to defining a `[` S3 method in R).
- `__getattr__`—The function invoked when accessing an attribute with `.` (equivalent to defining a `$` S3 method in R).
- `__iter__` and `__next__`—Functions invoked by `for`.

- `__call__`—Invoked when a class instance is called like a function (e.g., `instance()`).
- `__bool__`—Invoked by `if` and `while` (equivalent to `as.logical()` in R, but returning only a scalar, not a vector).
- `__repr__` and `__str__`—Functions invoked for formatting and pretty printing (akin to `format()`, `dput()`, and `print()` methods in R).
- `__enter__` and `__exit__`—Functions invoked by `with`.
- Many built-in Python functions are just sugar for invoking the dunder. For example, calling `repr(x)` is identical to `x.__repr__()` (see https://docs.python.org/3/library/functions.html). Other built-ins that are just sugar for invoking the dunder include `next()`, `iter()`, `str()`, `list()`, `dict()`, `bool()`, `dir()`, `hash()`, and more!

A.5.2 Iterators, revisited

Now that we have the basics of `class`, it's time to revisit iterators. First, some terminology:

- *iterable*—Something that can be iterated over. Concretely, a class that defines an `__iter__` method, whose job is to return an *iterator*.
- *iterator*—Something that iterates. Concretely, a class that defines a `__next__` method, whose job is to return the next element each time it is called, and then raise a `StopIteration` exception once it's exhausted. It's common to see classes that are both iterables and iterators, where the `__iter__` method is just a stub that returns `self`. Here is a custom iterable/iterator implementation of Python's `range()` (similar to `seq()` in R):

```
class MyRange:
  def __init__(self, start, end):
    self.start = start
    self.end = end

  def __iter__(self):
    self._index = self.start - 1    <- Reset our counter.
    return self

  def __next__(self):
    if self._index < self.end:
      self._index += 1    <- Increment by 1.
      return self._index
    else:
      raise StopIteration

for x in MyRange(1, 3):
  print(x)
```

```
1
2
3
```

Manually doing what `for` does:

```
r = MyRange(1, 3)
it = iter(r)
next(it)
```

```
1
```

```
next(it)
```

```
2
```

```
next(it)
```

```
3
```

```
next(it)
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): StopIteration
```

A.6 Defining generators with yield

Generators are special Python functions that contain one or more `yield` statements. As soon as `yield` is included in a code block passed to `def`, the semantics change substantially. You're no longer defining a mere function, but a generator constructor! In turn, calling a generator constructor creates a generator object, which is just another type of iterator. Here is an example:

```
def my_generator_constructor():
  yield 1
  yield 2
  yield 3
```

At first glance, it presents like a regular function:

```
my_generator_constructor
```

```
<function my_generator_constructor at 0x7f5e30fab670>
```

```
type(my_generator_constructor)
```

```
<class 'function'>
```

But calling it returns something special, a *generator object*:

```
my_generator = my_generator_constructor()
my_generator
```

```
<generator object my_generator_constructor at 0x7f5e3ca52820>
```

```
type(my_generator)
```

```
<class 'generator'>
```

The generator object is both an iterable and an iterator. Its `__iter__` method is just a stub that returns `self`:

```
iter(my_generator) == my_generator == my_generator.__iter__()
```

```
True
```

Step through it like any other iterator:

```
next(my_generator)
```

```
1
```

```
my_generator.__next__()
```

next(x) is just sugar for calling the dunder x.__next__().

```
2
```

```
next(my_generator)
```

```
3
```

```
next(my_generator)
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): StopIteration
```

Encountering `yield` is like hitting the pause button on a functions execution: it preserves the state of everything in the function body and returns control to whatever is iterating over the generator object. Calling `next()` on the generator object resumes execution of the function body until the next `yield` is encountered or the function finishes. You can create generators in R with `coro::generator()`.

A.7 Iteration closing remarks

Iteration is deeply baked into the Python language, and R users may be surprised by how things in Python are iterable, iterators, or powered by the iterator protocol under the hood. For example, the built-in `map()` (equivalent to R's `lapply()`) yields an iterator, not a list. Similarly, a tuple comprehension like `(elem for elem in x)` produces an iterator. Most features dealing with files are iterators.

Any time you find an iterator inconvenient, you can materialize all the elements into a list using the Python built-in `list()`, or `reticulate::iterate()` in R. Also, if you like the readability of `for`, you can utilize similar semantics to Python's `for` using `coro::loop()`.

A.8 import and modules

In R, authors can bundle their code into shareable extensions called R packages, and R users can access objects from R packages via `library()` or `::`. In Python, authors bundle code into *modules*, and users access modules using `import`. Consider the line:

```
import numpy
```

This statement has Python go out to the filesystem, find an installed Python module named numpy, load it (commonly meaning: evaluate its `__init__.py` file and construct a `module` type object), and bind it to the symbol `numpy`. The closest equivalent to this in R might be:

```
dplyr <- loadNamespace("dplyr")
```

A.8.1 Where are modules found?

In Python, the filesystem locations where modules are searched can be accessed (and modified) from the list found at `sys.path`. This is Python's equivalent to R's `.libPaths()`. `sys.path` will typically contain paths to the current working directory, the Python installation which contains the built-in standard library, administrator-installed modules, user-installed modules, values from environment variables like `PYTHONPATH`, and any modifications made directly to `sys.path` by other code in the current Python session (though this is relatively uncommon in practice):

```
import sys
sys.path

['',
'/home/tomasz/.pyenv/versions/3.9.6/bin',
'/home/tomasz/.pyenv/versions/3.9.6/lib/python39.zip',
'/home/tomasz/.pyenv/versions/3.9.6/lib/python3.9',
'/home/tomasz/.pyenv/versions/3.9.6/lib/python3.9/lib-dynload',
'/home/tomasz/.virtualenvs/r-reticulate/lib/python3.9/site-packages',
'/home/tomasz/opt/R-4.1.2/lib/R/site-library/reticulate/python',
'/home/tomasz/.virtualenvs/r-reticulate/lib/python39.zip',
'/home/tomasz/.virtualenvs/r-reticulate/lib/python3.9',
'/home/tomasz/.virtualenvs/r-reticulate/lib/python3.9/lib-dynload']
```

The current directory is typically on the search path for modules.

Python standard library and built-ins

reticulate shims

More standard library and built-ins, this time from the virtualenv

Additional installed Python packages (e.g., via pip)

You can inspect where a module was loaded from by accessing the dunder `__path__` or `__file__` (especially useful when troubleshooting installation issues):

```
import os
os.__file__

'/home/tomasz/.pyenv/versions/3.9.6/lib/python3.9/os.py'
```

The os module is defined here. It's just a regular text file; take a glance!

```
numpy.__path__

['/home/tomasz/.virtualenvs/r-reticulate/lib/python3.9/site-packages/numpy']
```

The numpy module we imported is defined here. It's a directory with lots of stuff; take a glance!

Once a module is loaded, you can access symbols from the module using `.` (equivalent to `::`, or maybe `$.environment`, in R):

```
numpy.abs(-1)
```

```
1
```

There is also special syntax for specifying the symbol a module is bound to upon import and for importing only some specific symbols:

```
import numpy
import numpy as np
np is numpy

from numpy import abs
abs is numpy.abs

from numpy import abs as abs2
abs2 is numpy.abs
```

Import and bind to symbol 'numpy'.

Import and bind to custom symbol 'np'.

Test for identicalness, similar to R identical(np, numpy). Returns True.

Import only numpy.abs, and bind it to abs.

True

Import only numpy.abs, and bind it to abs2.

True

If you're looking for the Python equivalent of R's `library()`, which makes all of a package's exported symbols available, it might be using `import` with a `*` wildcard, though it's relatively uncommon to do so. The `*` wildcard will expand to include all the symbols in module, or all the symbols listed in `__all__`, if it is defined:

```
from numpy import *
```

Python doesn't make a distinction like R does between package exported and internal symbols. In Python, all module symbols are equal, though there is the naming convention that intended-to-be-internal symbols are prefixed with a single leading underscore. (Two leading underscores invoke an advanced language feature called "name mangling," which is outside the scope of this introduction.)

If you're looking for the R equivalent to Python's `import` syntax, you can use `envir::import_from()` like this:

```
library(envir)
import_from(keras::keras$applications$efficientnet,
            decode_predictions, preprocess_input,
            new_model = EfficientNetB4)

model <- new_model(include_top = TRUE, weights='imagenet')

predictions <- input_data %>%
  preprocess_input() %>%
  predict(model, .) %>%
  decode_predictions()
```

A.9 Integers and floats

R users generally don't need to be aware of the difference between integers and floating-point numbers, but that's not the case in Python. If this is your first exposure to numeric data types, here are the essentials:

- Integer types can represent only whole numbers like `2` or `3`, not floating-point numbers like `2.3`.
- Floating-point types can represent any number, but with some degree of imprecision.

In R, writing a bare literal number like `3` produces a floating-point type, whereas in Python, it produces an integer. You can produce an integer literal in R by appending an `L`, as in `3L`. Many Python functions expect integers and will signal an error when provided a float. For example, say we have a Python function that expects an integer:

```
def a_strict_Python_function(x):
  assert isinstance(x, int), "x is not an int"
  print("Yay! x was an int")
```

When calling it from R, you must be sure to call it with an integer:

```
library(reticulate)
py$a_strict_Python_function(3)        Error: "AssertionError:
                                       x is not an int"

py$a_strict_Python_function(3L)
py$a_strict_Python_function(as.integer(3))     Success
```

A.10 What about R vectors?

R is a language designed for numerical computing first. Numeric vector data types are baked deep into the R language, to the point that the language doesn't even distinguish scalars from vectors. By comparison, numerical computing capabilities in Python are generally provided by third-party packages (*modules*, in Python parlance).

In Python, the numpy module is most commonly used to handle contiguous arrays of data. The closest equivalent to an R numeric vector is a 1D NumPy array, or sometimes, a list of scalar numbers (some Pythonistas might argue for `array.array()` here, but that's so rarely encountered in actual Python code we don't mention it further).

NumPy arrays are very similar to TensorFlow tensors. For example, they share the same broadcasting semantics and very similar indexing behavior. The NumPy API is extensive, and teaching the full NumPy interface is beyond the scope of this primer. However, it's worth pointing out some potential tripping hazards for users accustomed to R arrays:

- When indexing into multidimensional NumPy arrays, trailing dimensions can be omitted and are implicitly treated as missing. The consequence is that iterating over arrays means iterating over the first dimension. For example, this iterates over the rows of a matrix:

```
import numpy as np
m = np.arange(12).reshape((3,4))
m
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
m[0, :]        <-- First row
```

```
array([0, 1, 2, 3])
```

```
m[0]           <-- Also first row
```

```
array([0, 1, 2, 3])
```

```
for row in m:
  print(row)
```

```
[0 1 2 3]
[4 5 6 7]
[ 8  9 10 11]
```

- Many NumPy operations modify the array in place! This is surprising to R users (and TensorFlow users), who are used to the convenience and safety of R's (and TensorFlow's) copy-on-modify semantics. Unfortunately, there is no simple scheme or naming convention you can rely on to quickly determine whether a particular method modifies in place or creates a new array copy. The only reliable way is to consult the documentation (see http://mng.bz/mORP), and conduct small experiments at the `reticulate::repl_python()`.

A.11 Decorators

Decorators are just functions that take a function as an argument and then typically return another function. Any function can be invoked as a decorator with the `@` syntax, which is just sugar for this simple action:

```
def my_decorator(func):
  func.x = "a decorator modified this function by adding an attribute `x`"
  return func

@my_decorator
def my_function(): pass


def my_function(): pass
my_function = my_decorator(my_function)    <-- @decorator is just fancy
                                               syntax for this line.
```

One decorator you might encounter frequently is `@property`, which automatically calls a class method when the attribute is accessed (similar to `makeActiveBinding()` in R):

```
from datetime import datetime
class MyClass:
  @property
  def a_property(self):
    return f"`a_property` was accessed at {datetime.now().strftime('%X')}"

instance = MyClass()
instance.a_property
```

```
'`a_property` was accessed at 10:01:53 AM'
```

You can translate Python's `@property` to R with `%<-active%` (or with `mark_active()`), like this:

```
import_from(glue, glue)
MyClass %py_class% {
  a_property %<-active% function()
    glue("`a_property` was accessed at {format(Sys.time(), '%X')}")
}

instance <- MyClass()
instance$a_property
```

```
[1] "`a_property` was accessed at 10:01:53 AM"
```

```
Sys.sleep(1)
instance$a_property
```

```
[1] "`a_property` was accessed at 10:01:54 AM"
```

A.12 with and context management

Any object that defines `__enter__` and `__exit__` methods implements the "context" protocol and can be passed to `with`. For example, here is a custom implementation of a context manager that temporarily changes the current working directory (equivalent to R's `withr::with_dir()`):

```
from os import getcwd, chdir

class wd_context:
  def __init__(self, wd):
    self.new_wd = wd

  def __enter__(self):
    self.original_wd = getcwd()
    chdir(self.new_wd)

  def __exit__(self, *args):
    chdir(self.original_wd)


getcwd()
```

__exit__ takes some additional argument that are often ignored.

```
'/home/tomasz/deep-learning-w-R-v2/manuscript'
```

```
with wd_context("/tmp"):
  print("in the context, wd is:", getcwd())
```

```
in the context, wd is: /tmp
```

```
getcwd()
```

```
'/home/tomasz/deep-learning-w-R-v2/manuscript'
```

A.13 Learning more

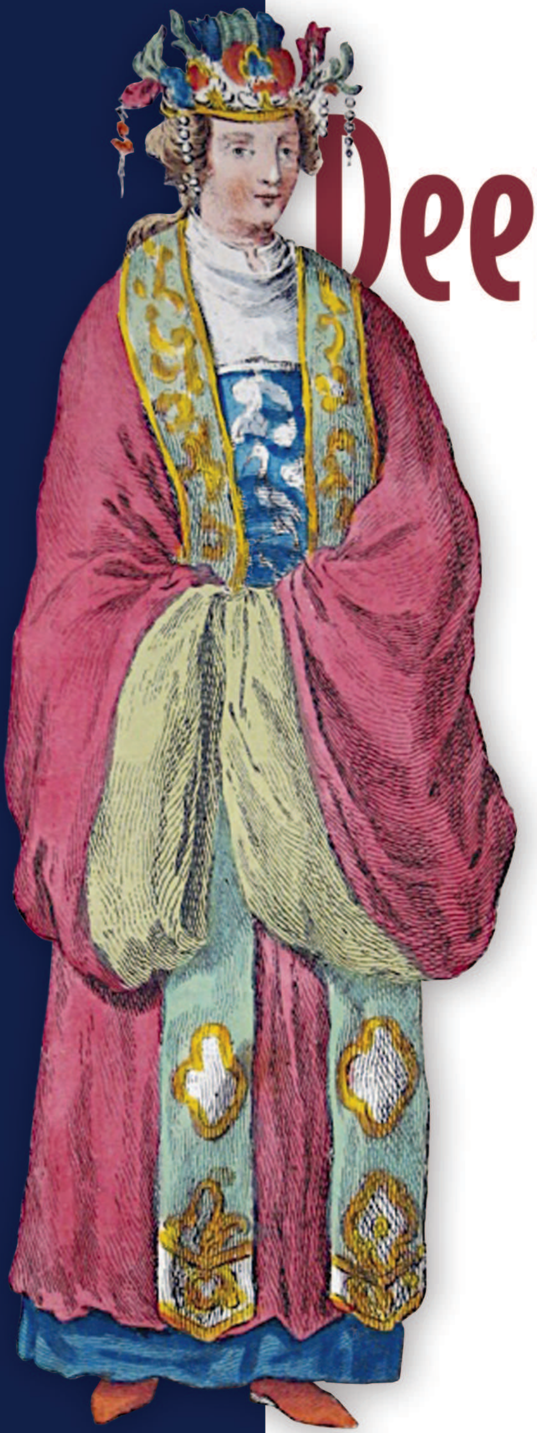
Hopefully, this short primer to Python has provided a good foundation for confidently reading Python documentation and code, and using Python modules from R via reticulate. Of course, there is much, much more to learn about Python. Googling questions about Python reliably brings up pages of results, but not always sorted in order of most useful. Blog posts and tutorials targeting beginners can be valuable, but remember that Python's official documentation is generally excellent, and it should be your first destination when you have questions:

- https://docs.Python.org/3/
- https://docs.Python.org/3/library/index.html

To learn Python more fully, the built-in official tutorial is also excellent and comprehensive (but does require a time commitment to get value out of it): https://docs.Python.org/3/tutorial/index.html.

Finally, don't forget to solidify your understanding by conducting small experiments at the `reticulate::repl_python()`.

Thank you for reading!



Deep Learning with R

SECOND EDITION

François Chollet
with Tomasz Kalinowski
and J. J. Allaire

MANNING

DEEP LEARNING / R

Deep Learning with R Second Edition

Chollet • Kalinowski • Allaire

Deep learning has become essential knowledge for data scientists, researchers, and software developers. The R language APIs for Keras and TensorFlow put deep learning within reach for all R users, even if they have no experience with advanced machine learning or neural networks. This book shows you how to get started on core DL tasks like computer vision, natural language processing, and more using R.

Deep Learning with R, Second Edition is a hands-on guide to deep learning using the R language. As you move through this book, you'll quickly lock in the foundational ideas of deep learning. The intuitive explanations, crisp illustrations, and clear examples guide you through core DL skills like image processing and text manipulation, and even advanced features like transformers. This revised and expanded new edition is adapted from *Deep Learning with Python, Second Edition* by François Chollet, the creator of the Keras library.

What's Inside

- Image classification and image segmentation
- Time series forecasting
- Text classification and machine translation
- Text generation, neural style transfer, and image generation

For readers with intermediate R skills. No previous experience with Keras, TensorFlow, or deep learning is required.

François Chollet is a software engineer at Google and creator of Keras. **Tomasz Kalinowski** is a software engineer at RStudio and maintainer of the Keras and Tensorflow R packages. **J.J. Allaire** is the founder of RStudio, and the author of the first edition of this book.

"A must-have for scientists and technicians who want to expand their knowledge."
—Fernando García Sedano
Grupo Epelsa

"Whether you are new to deep learning or wanting to expand your applications in R, there is no better guide."
—Michael Petrey
Boxplot Analytics

"The clear illustrations and insightful examples are helpful to anybody, from beginners to experienced deep learning practitioners."
—Edward Lee, Yale University

"Outstandingly well written."
—Shahnawaz Ali
King's College London

Free eBook
See first page

Register this print book to get free access to all ebook formats.
Visit https://www.manning.com/freebook

MANNING

ISBN-13: 978-1-63343-984-9
90000
9 781633 439849