Types in R & Python

Lecture 02

Dr. Colin Rundel

```py
import math
```

Type systems

Dynamic typing

Both R and Python are *dynamically typed* languages - the type of a value is determined at runtime and a variable (name) is not restricted to a single type.

```r
x = 1
typeof(x)
```

```
[1] "double"
```

```r
x = "one"
typeof(x)
```

```
[1] "character"
```

```py
x = 1
type(x)
```

```
<class 'int'>
```

```py
x = "one"
type(x)
```

```
<class 'str'>
```

Where the languages differ is in their fundamental unit of data,

- in R (almost) everything is a *vector*
- in Python individual values are separate *objects* that can be grouped into *container* objects

Dynamic describes when types are checked; it does not mean that values lack types or that every operation between

Vectors in R

The fundamental building block for data in R is the vector—a collection of related values.

R has two types of vectors:

- atomic vectors (*vectors*) — homogeneous collections with a single underlying type (e.g. `logical`, `integer`, `double`, or `character`).
- generic vectors (*lists*) — heterogeneous collections that can contain any R object, including other lists, allowing hierarchical or tree-like structures.

There are no scalars in R - a single value like `1` is just a vector of length 1.

```r
length(1)
```

```
[1] 1
```

```r
length(c(1, 2, 3))
```

```
[1] 3
```

Objects in Python

Python is built around an object-oriented class system—*every* value is an object with a type, including numbers, strings, and containers.

```py
1
```

```
1
```

```py
type(1)
```

```
<class 'int'>
```

```py
"one"
```

```
'one'
```

```py
type("one")
```

```
<class 'str'>
```

```py
[1, 2, 3]
```

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

```py
type([1, 2, 3])
```

```
<class 'list'>
```

Individual objects can be collected into general-purpose *containers* such as `list`, which can be heterogeneous and nested. Base Python has no direct equivalent to R's atomic vectors - that role is filled by `numpy` arrays (more on these next week).

Basic types

R's atomic types

R has six atomic vector types, we can check the type of any object using `typeof()`, `mode()` (a higher level grouping of types), or `class()`.

`typeof()`	`mode()`	`class()`	Example
logical	logical	logical	`TRUE`
double	numeric	numeric	`1.5`, `2`
integer	numeric	integer	`1L`, `1:3`
character	character	character	`"hello"`
complex	complex	complex	`1+2i`
raw	raw	raw	`as.raw(255)`

There are additional types in R, e.g. `list`, `closure`, `environment`, etc. which we will see in the coming lectures. See `?typeof` for more information.

Python's basic types

Python's common built-in value types are similar, but each value is an individual object rather than a vector.

Type	`type()`	Example
boolean	`bool`	`True`
integer	`int`	`1`
float	`float`	`1.5`, `2.0`
complex	`complex`	`1+2j`
string	`str`	`"hello"`
none	`NoneType`	`None`

There is also `bytes` (similar to R's `raw`), as well as container types such as `list`, `dict`, and `set`. We will introduce lists later

Logical / boolean values

```r
typeof(TRUE)
```

```
[1] "logical"
```

```r
typeof(FALSE)
```

```
[1] "logical"
```

```r
TRUE + TRUE
```

```
[1] 2
```

```r
sum(c(TRUE, FALSE, TRUE))
```

```
[1] 2
```

```python
type(True)
```

```
<class 'bool'>
```

```python
type(False)
```

```
<class 'bool'>
```

```python
True + True
```

```
2
```

```python
sum([True, False, True])
```

```
2
```

In both languages boolean values behave like the integers `1` and `0` when used in arithmetic. In Python this is explicit - `bool` is a subclass of `int`.

R lets you use `T` and `F` as shortcuts for `TRUE` and `FALSE` - this is bad practice as these are just global variables which can be

Numeric values

```r
typeof(1.5)
```

```
[1] "double"
```

```r
typeof(7)
```

```
[1] "double"
```

```r
typeof(7L)
```

```
[1] "integer"
```

```r
typeof(1:3)
```

```
[1] "integer"
```

```python
type(1.5)
```

```
<class 'float'>
```

```python
type(7)
```

```
<class 'int'>
```

```python
type(7.0)
```

```
<class 'float'>
```

```python
type(7.)
```

```
<class 'float'>
```

The *default* numeric type differs between the languages - a bare literal like `7` is a `double` in R but an `int` in Python. In R, integer literals require an `L` suffix, in Python floats require a decimal point (or an exponent, e.g. `1e3`).

Integers

R integers are 32-bit and overflow to `NA` (with a warning), Python integers have arbitrary precision.

```r
.Machine$integer.max
```

```
[1] 2147483647
```

```r
.Machine$integer.max + 1L
```

```
Warning in .Machine$integer.max + 1L: NAs
[1] NA
```

```r
.Machine$integer.max + 1
```

```
[1] 2147483648
```

```r
2^100
```

```
[1] 1.267651e+30
```

```python
2**31 - 1
```

```
2147483647
```

```python
2**31
```

```
2147483648
```

```python
2**100
```

```
1267650600228229401496703205376
```

In practice this rarely matters in R since the default numeric type is `double`, but it is one reason why R's `integer` type is used sparingly (e.g. for indexing and counts). Note that `numpy` and `pandas` behave more like R here, as they use fixed width integers (e.g. `int64`).

Floating-point precision

R's `double` and Python's `float` use binary floating-point representations, so many decimal values cannot be represented exactly.

```r
0.1 + 0.2
```

```
[1] 0.3
```

```r
0.1 + 0.2 == 0.3
```

```
[1] FALSE
```

```r
all.equal(0.1 + 0.2, 0.3)
```

```
[1] TRUE
```

```python
0.1 + 0.2
```

```
0.30000000000000004
```

```python
0.1 + 0.2 == 0.3
```

```
False
```

```python
import math
math.isclose(0.1 + 0.2, 0.3)
```

```
True
```

Complex numbers

Both languages have a built-in complex type - the main difference is the literal syntax, R uses `i` for the imaginary unit while Python uses `j` (the engineering convention).

```r
z = 1+2i; typeof(z)
```

```
[1] "complex"
```

```r
c(Re(z), Im(z), Mod(z))
```

```
[1] 1.000000 2.000000 2.236068
```

```python
z = 1+2j; type(z)
```

```
<class 'complex'>
```

```python
(z.real, z.imag, abs(z))
```

```
(1.0, 2.0, 2.23606797749979)
```

With a real negative input, R's `sqrt()` returns `NaN` while Python's `math.sqrt()` raises an error. R requires complex input, whereas Python's `**` operator promotes the result to complex.

```r
sqrt(-1)
```

```
Warning in sqrt(-1): NaNs produced
[1] NaN
```

```r
sqrt(-1+0i)
```

```
[1] 0+1i
```

```r
(-1)^0.5
```

```
[1] NaN
```

```python
math.sqrt(-1)
```

```
ValueError: expected a nonnegative input, got -1
```

```python
sqrt(-1+0j)
```

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

```python
(-1) ** 0.5
```

```
(6.123233995736766e-17+1j)
```

Numeric promotion / coercion

Both languages *promote* to the more "general" type when mixing numeric types (`logical` / `bool` → `integer` / `int` → `double` / `float` → `complex`),

```r
typeof(1L + 1.5)
```

```
[1] "double"
```

```py
type(1 + 1.5)
```

```
<class 'float'>
```

```r
typeof(TRUE + 1L)
```

```
[1] "integer"
```

```py
type(True + 1)
```

```
<class 'int'>
```

```r
typeof(1L + 2i)
```

```
[1] "complex"
```

```py
type(1 + 2j)
```

```
<class 'complex'>
```

For integer inputs, division (`/`) returns a `double` / `float` rather than an integer in both languages,

```r
4L / 2L
```

```
[1] 2
```

```py
4 / 2
```

```
2.0
```

```r
typeof(4L / 2L)
```

```
[1] "double"
```

```py
type(4 / 2)
```

```
<class 'float'>
```

Type predicates

```r
is.integer(1)
```

```
[1] FALSE
```

```r
is.integer(1L)
```

```
[1] TRUE
```

```r
is.double(1)
```

```
[1] TRUE
```

```r
is.numeric(1)
```

```
[1] TRUE
```

```r
is.numeric(1L)
```

```
[1] TRUE
```

```r
is.character("1")
```

```
[1] TRUE
```

```python
isinstance(1, int)
```

```
True
```

```python
isinstance(1.0, int)
```

```
False
```

```python
isinstance(1.0, float)
```

```
True
```

```python
isinstance(True, int)
```

```
True
```

```python
isinstance("1", str)
```

```
True
```

`is.numeric()` checks the *mode* (`double` or `integer`) while the others check the *type*. Python's `isinstance()` respects class inheritance (hence `True` is an `int`).

Strings

String literals

```r
typeof("hello")
```

```
[1] "character"
```

```r
typeof('world')
```

```
[1] "character"
```

```r
"abc'123"
```

```
[1] "abc'123"
```

```r
'abc"123'
```

```
[1] "abc\"123"
```

```r
"abc\"123"
```

```
[1] "abc\"123"
```

```python
type("hello")
```

```
<class 'str'>
```

```python
type('world')
```

```
<class 'str'>
```

```python
"abc'123"
```

```
"abc'123"
```

```python
'abc"123'
```

```
'abc"123'
```

```python
"abc\"123"
```

```
'abc"123'
```

In both languages single or double quotes are fine (the opening and closing quote must match). Quote characters can be included by escaping or by using the non-matching quote.

Multi-line strings

R strings can contain newlines directly (or via `\n`), in Python a literal must be triple quoted (`"""` or `'''`) to span multiple lines.

```r
x = "line one
line 'two'"
x
```

```
[1] "line one\nline 'two'"
```

```r
cat(x)
```

```
line one
line 'two'
```

```python
x = """line one
line 'two'"""
x
```

```
"line one\nline 'two'"
```

```python
print(x)
```

```
line one
line 'two'
```

String interpolation

Python's f-strings (3.6+) evaluate arbitrary expressions inside `{}`. R has no built-in equivalent - `paste()` / `paste0()` are the base approach and the `glue` package provides similar `{}` interpolation.

R
```r
x = c(1, 2, 3)
paste0("x has ", length(x), " elements")
```

```
[1] "x has 3 elements"
```

R
```r
glue::glue("x has {length(x)} elements")
```

```
x has 3 elements
```

R
```r
glue::glue("From {min(x)} to {max(x)}")
```

```
From 1 to 3
```

py
```python
x = [1, 2, 3]
"x has " + str(len(x)) + " elements"
```

```
'x has 3 elements'
```

py
```python
f"x has {len(x)} elements"
```

```
'x has 3 elements'
```

py
```python
f"From {min(x)} to {max(x)}"
```

```
'From 1 to 3'
```

Coercion

Atomic vectors are homogeneous

R's atomic vectors must contain values of a single type, so combining different types with `c()` forces *coercion* to the most general type present,

```r
c(1, "Hello")
```

```
[1] "1"     "Hello"
```

```r
c(1.2, 3L)
```

```
[1] 1.2 3.0
```

```r
c(FALSE, 3L)
```

```
[1] 0 3
```

```r
c(FALSE, "Hello")
```

```
[1] "FALSE" "Hello"
```

Atomic vectors have flat underlying storage—nested calls to `c()` combine their contents rather than create a nested structure,

```r
c(1, c(2, c(3)))
```

```
[1] 1 2 3
```

Base Python has no direct counterpart. Lists can mix types and be nested, like R's lists; `numpy` arrays are closer because they are homogeneous and choose a common data type. We will see these later.

Implicit coercion in R

R is quite liberal about coercion - builtin operators and functions (e.g. `+`, `&`, `log()`, etc.) will generally attempt to coerce values to an appropriate type for the given operation (numeric for math, logical for logical, etc.)

```r
3.1 + 1L
```

```
[1] 4.1
```

```r
5 + FALSE
```

```
[1] 5
```

```r
log(TRUE)
```

```
[1] 0
```

```r
TRUE & 7
```

```
[1] TRUE
```

```r
FALSE | !5
```

```
[1] FALSE
```

```r
TRUE == "TRUE"
```

```
[1] TRUE
```

```r
1 == "1"
```

```
[1] TRUE
```

Arithmetic with strings is the exception, R will not coerce a character value to a number for math,

```r
"1" + 1
```

```
Error in `"1" + 1`:
! non-numeric argument to binary operator
```

Implicit coercion in Python

Python is more conservative: it promotes values within the numeric types (`bool` → `int` → `float` → `complex`) but generally does not implicitly convert between strings and numbers. Equality comparisons between these types return `False`, while unsupported ordering and arithmetic operations raise an error.

```py
1 True + 1
```

```
2
```

```py
1 5 + False
```

```
5
```

```py
1 1 == True
```

```
True
```

```py
1 (1+0j) == 1
```

```
True
```

```py
1 1 == "1"
```

```
False
```

```py
1 "abc" > 5
```

```
TypeError: '>' not supported between insta
```

```py
1 "abc" + 5
```

```
TypeError: can only concatenate str (not "
```

```py
1 "abc" + str(5)
```

```
'abc5'
```

Operator overloading in Python

Do be aware that Python has a number of useful / idiosyncratic overloads for some of the basic operators - depending on the types involved the same operator can do very different things.

```py
"abc" + "def"
```

```
'abcdef'
```

```py
"abc" * 3
```

```
'abcabcabc'
```

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

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

```py
[1, 2] * 2
```

```
[1, 2, 1, 2]
```

```py
"abc" ** 2
```

```
TypeError: unsupported operand type(s) for
```

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

```
TypeError: can't multiply sequence by non-
```

Explicit coercion

R uses the `as.*()` family of functions for explicit conversion, Python uses the type constructors (`int()`, `float()`, `str()`, `bool()`, etc.),

```r
as.logical(5.2)
```

```
[1] TRUE
```

```r
as.integer(pi)
```

```
[1] 3
```

```r
as.double("7.2")
```

```
[1] 7.2
```

```r
as.integer("2.1")
```

```
[1] 2
```

```r
as.integer("one")
```

```
Warning: NAs introduced by coercion
[1] NA
```

```python
bool(5.2)
```

```
True
```

```python
int(3.14159)
```

```
3
```

```python
float("7.2")
```

```
7.2
```

```python
int("2.1")
```

```
ValueError: invalid literal for int() with base
```

Coercion to logical / bool

```r
as.logical(0)
```

```
[1] FALSE
```

```r
as.logical("TRUE")
```

```
[1] TRUE
```

```r
as.logical("T")
```

```
[1] TRUE
```

```r
as.logical("yes")
```

```
[1] NA
```

```r
as.logical("")
```

```
[1] NA
```

```py
bool(0)
```

```
False
```

```py
bool("False")
```

```
True
```

```py
bool("")
```

```
False
```

```py
bool([])
```

```
False
```

```py
bool(None)
```

```
False
```

Python uses the idea of *truthiness* (`0`, `0.0`, `""`, `None`, and empty containers are *falsy*, almost everything else is *truthy*) throughout the language, e.g. in conditionals. For character input, R recognizes only `"TRUE"`, `"true"`, `"True"`, `"T"` (and their `FALSE` counterparts); other strings become `NA`.

Missing & special values

Missing values in R

R uses `NA` to represent missing values in its data structures.

What may not be obvious is that there are different `NA`s for the different atomic types.

```r
NA
```

```
[1] NA
```

```r
typeof(NA)
```

```
[1] "logical"
```

```r
typeof(NA_character_)
```

```
[1] "character"
```

```r
NA+1
```

```
[1] NA
```

```r
typeof(NA+1)
```

```
[1] "double"
```

```r
typeof(NA_real_)
```

```
[1] "double"
```

```r
NA+1L
```

```
[1] NA
```

```r
typeof(NA+1L)
```

```
[1] "integer"
```

```r
typeof(NA_integer_)
```

```
[1] "integer"
```

```r
c(NA,"")
```

```
[1] NA ""
```

```r
typeof(c(NA,""))
```

```
[1] "character"
```

```r
typeof(NA_complex_)
```

```
[1] "complex"
```

NA stickiness

As `NA`s represent missing values (most) calculations using them return a missing value.

```r
1 + NA
```

```
[1] NA
```

```r
1 / NA
```

```
[1] NA
```

```r
NA * 5
```

```
[1] NA
```

```r
sqrt(NA)
```

```
[1] NA
```

```r
3^NA
```

```
[1] NA
```

```r
sum(c(1, 2, 3, NA))
```

```
[1] NA
```

Aggregation / summarization functions (e.g. `sum()`, `mean()`, `sd()`, etc.) will often have an `na.rm` argument which *removes* the missing values from the calculation.

```r
sum(c(1, 2, 3, NA), na.rm = TRUE)
```

```
[1] 6
```

```r
mean(c(1, 2, 3, NA), na.rm = TRUE)
```

```
[1] 2
```

NAs are not always sticky

A useful mental model for NAs is to consider them as an unknown value that could take any of the possible values for a given type.

For numbers or characters this isn't helpful, but for a logical value it must either be TRUE or FALSE which is relevant for certain calculations.

```r
TRUE & NA
```

```
[1] NA
```

```r
TRUE | NA
```

```
[1] TRUE
```

```r
FALSE & NA
```

```
[1] FALSE
```

```r
FALSE | NA
```

```
[1] NA
```

Special values (double)

R also has the non-finite values defined by the IEEE floating point standard (these are not unique to R): `NaN` (not a number), `Inf`, and `-Inf`.

```r
-1 / 0
```

```
[1] -Inf
```

```r
0 / 0
```

```
[1] NaN
```

```r
1/0 + 1/0
```

```
[1] Inf
```

```r
Inf - Inf
```

```
[1] NaN
```

```r
NaN / NA
```

```
[1] NA
```

```r
NaN * NA
```

```
[1] NA
```

These are all `double` values, but their coercion behavior is not the same as other doubles,

```r
typeof(Inf)
```

```
[1] "double"
```

```r
typeof(NaN)
```

```
[1] "double"
```

```r
as.integer(Inf)
```

```
Warning: NAs introduced by coercion to integer r
[1] NA
```

```r
as.integer(NaN)
```

```
[1] NA
```

Testing for NA, NaN, and Inf

```r
is.na(NA)
```

```
[1] TRUE
```

```r
is.nan(NA)
```

```
[1] FALSE
```

```r
is.finite(NA)
```

```
[1] FALSE
```

```r
is.na(NaN)
```

```
[1] TRUE
```

```r
is.nan(NaN)
```

```
[1] TRUE
```

```r
is.finite(NaN)
```

```
[1] FALSE
```

```r
is.na(Inf)
```

```
[1] FALSE
```

```r
is.nan(Inf)
```

```
[1] FALSE
```

```r
is.finite(Inf)
```

```
[1] FALSE
```

Note that `is.na()` is `TRUE` for `NaN` but `is.nan()` is `FALSE` for `NA`. Comparisons involving `NA` or `NaN` always return `NA` so these predicates must be used instead of `==`.

```r
NA == NA
```

```
[1] NA
```

```r
NaN == NaN
```

```
[1] NA
```

```r
c(1, NA, 3) == NA
```

```
[1] NA NA NA
```

NULL in R

`NULL` represents the absence of an object or value. Unlike `NA`, it has length zero and is removed when combined with an atomic vector.

```r
typeof(NULL)
```

```
[1] "NULL"
```

```r
length(NULL)
```

```
[1] 0
```

```r
is.null(NULL)
```

```
[1] TRUE
```

```r
c(1, NULL, 2)
```

```
[1] 1 2
```

```r
list(1, NULL, 2)
```

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

[[2]]
NULL

[[3]]
[1] 2
```

`None` in Python

Base Python has no built-in missing-value type. `None` is a singleton object (of type `NoneType`) used to represent "no value," making it closer to R's `NULL` than to `NA`.

```py
print(None)
```

```
None
```

```py
type(None)
```

```
<class 'NoneType'>
```

```py
x = None
x is None
```

```
True
```

```py
None + 1
```

```
TypeError: unsupported operand type(s) for
```

```py
[1, None, "a"]
```

```
[1, None, 'a']
```

Unlike `NA`, `None` does not propagate through calculations (it raises an error) and it has no type-specific variants.

Because `None` is a singleton, test for it with `is None` or `is not None` rather than `==` or `!=`. Missing values are instead

Non-finite values in Python

`nan` and `inf` are available (as `floats`) via `float()` or the `math` module. Note that division by zero is *always* an error in Python (even for floats) rather than producing `inf`.

```py
1 / 0
```

```
ZeroDivisionError: division by
```

```py
float("inf")
```

```
inf
```

```py
float("nan")
```

```
nan
```

```py
math.inf
```

```
inf
```

```py
math.nan
```

```
nan
```

```py
math.inf - math.inf
```

```
nan
```

```py
math.isnan(math.nan)
```

```
True
```

```py
math.isinf(-math.inf)
```

```
True
```

```py
math.nan == math.nan
```

```
False
```

Testing is done with `math.isnan()`, `math.isinf()`, and `math.isfinite()` and, as in R, `nan` is not equal to itself (or anything else).

Exercise 1

Part 1

What is the type of the following R vectors? Explain why they have that type.

```r
c(1, NA+1L, "C")
c(1L / 0, NA)
c(1:3, 5)
c(3L, NaN+1L)
c(NA, TRUE)
c(1L, NULL, NA)
```

Part 2

Without running the code, what is the value (and type) of each of the following Python expressions?

```python
True + 1.5
"1" * 3
int("3") + float("1.5")
bool("False")
int(True) / True
```

Part 3

Considering R's four most commonly used atomic types (`logical`, `integer`, `double`, and `character`), what is the implicit conversion hierarchy from highest to lowest priority? How does this compare to Python's numeric promotion rules?

Python lists

Lists

Python lists are *ordered*, *mutable* sequences that can contain objects of different types (very similar to lists in R, which we will see in a few lectures).

```py
[0, 1, 1, 0]
```

```
[0, 1, 1, 0]
```

```py
[0, True, "abc"]
```

```
[0, True, 'abc']
```

```py
[0, [1, 2], [3, [4]]]
```

```
[0, [1, 2], [3, [4]]]
```

```py
x = [0, 1, 1, 0]
type(x)
```

```
<class 'list'>
```

```py
len(x)
```

```
4
```

```py
2 in x
```

```
False
```

```py
x + [3, 4, 5]
```

```
[0, 1, 1, 0, 3, 4, 5]
```

Indexing

Elements are accessed using `[]` with **0-based indexing**, negative indexes count from the end, and ranges of values can be specified using slices (`start:stop` where `stop` is *exclusive*).

```py
x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```py
x[0]
```

```
1
```

```py
x[3]
```

```
4
```

```py
x[-1]
```

```
9
```

```py
x[0:3]
```

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

```py
x[3:]
```

```
[4, 5, 6, 7, 8, 9]
```

```py
x[-3:]
```

```
[7, 8, 9]
```

Mutability

Since lists are mutable the stored values can be changed, removed, or added to (in place).

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

```py
x[0] = -1
x
```

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

```py
del x[0]
x
```

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

```py
x.append(7)
x
```

```
[2, 3, 4, 5, 7]
```

```py
x.insert(1, -5)
x
```

```
[2, -5, 3, 4, 5, 7]
```

```py
x.pop()
```

```
7
```

```py
x
```

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

Assignment & identity

Assignment & copies

Ordinary R vectors use copy-on-modify semantics: after `y = x`, modifying `y` does not change `x`. Reference-style objects such as environments are exceptions.

In Python, assignment binds a *name* to an *object*: both names refer to the same object, so modifying a mutable object through one name is visible through the other.

R

```r
x = c(1, 2, 3)
y = x
y[1] = 100
```

R

```r
x
```

```
[1] 1 2 3
```

R

```r
y
```

```
[1] 100   2   3
```

py

```python
x = [1, 2, 3]
y = x
y[0] = 100
```

py

```python
x
```

```
[100, 2, 3]
```

py

```python
y
```

```
[100, 2, 3]
```

Shallow copies

To make an independent copy of a Python list, use `.copy()`:

```py
x = [1, 2, 3]
y = x.copy()
y[0] = 100
x
```

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

A list's `.copy()` method copies the outer list, but nested mutable objects remain shared.

```py
x = [[1, 2], [3, 4]]
y = x.copy()

y[0][0] = 100
x
```

```
[[100, 2], [3, 4]]
```

```py
y
```

```
[[100, 2], [3, 4]]
```

Identity vs equality

Because multiple Python names can refer to the same object, Python distinguishes between *identity* (`is` - are these the same object?) and *equality* (`==` - do they have the same value?). The `id()` function returns an integer representing an object's unique identity.

```py
x = [1, 2, 3]
y = x
z = x.copy()
```

```py
x is y
```

```
True
```

```py
x is z
```

```
False
```

```py
x == z
```

```
True
```

```py
id(x)
```

```
4563224640
```

```py
id(y)
```

```
4563224640
```

```py
id(z)
```

```
4563226112
```

R's `identical()` tests strict value equality (including type and attributes), not object identity. For example, `identical(1L, 1)` is `FALSE` while `1L == 1` is `TRUE`. With ordinary copy-on-modify objects, identity rarely matters in R.

Comparing R & Python

Scalars vs vectors

For numeric data, the biggest conceptual difference is that R represents even a single atomic value as a vector, while base Python distinguishes individual numeric values from containers. This affects how arithmetic on collections behaves,

R

```r
x = c(1, 2, 3)
length(x)
```

```
[1] 3
```

R

```r
x * 2
```

```
[1] 2 4 6
```

R

```r
x + x
```

```
[1] 2 4 6
```

R

```r
length(1)
```

```
[1] 1
```

py

```python
x = [1, 2, 3]
len(x)
```

```
3
```

py

```python
x * 2
```

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

py

```python
x + x
```

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

py

```python
len(1)
```

```
TypeError: object of type 'int' has no len
```

Next week we will see how `numpy` arrays bring R-style vectorized operations to Python.

Type summary

Concept	R	Python
boolean	`logical` (`TRUE`, `FALSE`)	`bool` (`True`, `False`)
integer	`integer` (`1L`)	`int` (`1`)
real	`double` (`1`, `1.5`)	`float` (`1.0`, `1.5`)
complex	`complex` (`1i`)	`complex` (`1j`)
text	`character` (`"a"`)	`str` (`"a"`)
type of	`typeof()`, `class()`, `is.*()`	`type()`, `isinstance()`
conversion	`as.integer()`, `as.character()`, …	`int()`, `str()`, …
missing	`NA` (typed)	library-specific sentinels
absent / null	`NULL`	`None`
non-finite	`Inf`, `-Inf`, `NaN`	`math.inf`, `-math.inf`, `math.nan`
containers	atomic vectors, lists	`list`, …

Operator summary

Operation	R	Python
exponent	`^`	`**`
integer division	`%/%`	`//`
modulus	`%%`	`%`
and / or / not	`&`, `|`, `!`	`and`, `or`, `not`
value equality	`==`, `identical()`	`==`
object identity	—	`is`
membership	`%in%`	`in`
string concat	`paste0()`	`+`
interpolation	`glue::glue()`	f-strings
raw strings	`r"(...)"`	`r"..."`

These are not exact equivalents: R's `&` and `|` are vectorized and evaluate both sides, whereas Python's `and` and `or` short-circuit and return one of their operands. R also has the short-circuit operators `&&` and `||`, which we will discuss with

Indexing

R uses **1-based** indexing and negative indexes *exclude* elements, Python uses **0-based** indexing and negative indexes count from the *end*.

R

```r
x = c("a", "b", "c", "d")
x[1]
```

```
[1] "a"
```

py

```python
x = ["a", "b", "c", "d"]
x[0]
```

```
'a'
```

R

```r
x[4]
```

```
[1] "d"
```

py

```python
x[3]
```

```
'd'
```

R

```r
x[-1]
```

```
[1] "b" "c" "d"
```

py

```python
x[-1]
```

```
'd'
```

R

```r
x[2:3]
```

```
[1] "b" "c"
```

py

```python
x[1:3]
```

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

Note that R's `2:3` *includes* both endpoints while Python's `1:3` slice *excludes* the stop value - both select the 2nd and 3rd elements here.

Exercise 2

Without running the code, predict the output (or error) of each of the following in *both* R and Python.

R
```r
x = c(1, 2, 3)
y = x
y[1] = "a"
x
y
```

py
```python
x = [1, 2, 3]
y = x
y[0] = "a"
x
y
```

R
```r
TRUE + TRUE == 2
"2" == 2
as.integer("2") + 1L
0.1 + 0.2 == 0.3
```

py
```python
True + True == 2
"2" == 2
int("2") + 1
0.1 + 0.2 == 0.3
```

R
```r
c(1, 2.5, NA)
sum(c(1, 2.5, NA))
```

py
```python
[1, 2.5, None]
sum([1, 2.5, None])
```

Takeaways

- R's atomic vectors are homogeneous and vectorized; base Python uses individual objects and general-purpose containers.
- Types determine which operations are valid and when coercion occurs.
- `NA`, `NULL`, `None`, and `nan` represent different ideas.
- R vectors usually copy on modify; Python mutable objects can be shared by multiple names.