Data types

Type	e.g.	C	R	Python
Integers	7, 49284	int	integer	int
Decimals	7.0, 12.56	float, double	numeric	float
Boolean	True, False	-	logical	bool
Strings	"a", "hello", "%"	char	character	str

Querying data types & data structures

Concept	R	Python
Data type	typeof(object)	type(element)
Data structure	class(object)	type(object)

In C, data types are defined ("declared") in the code explicitly.

```
> x = matrix(1.2, 2, 3)
> class(x)
[1] "matrix" "array"
> typeof(x)
[1] "double"
> y = 3L
> typeof(y)
[1] "integer"
```

R

```
>>> x = [1, 2, 3, 4]
>>> type(x)
list
>>> type(x[1])
int
>>> y = 3.2
>>> type(y)
float
```

Py

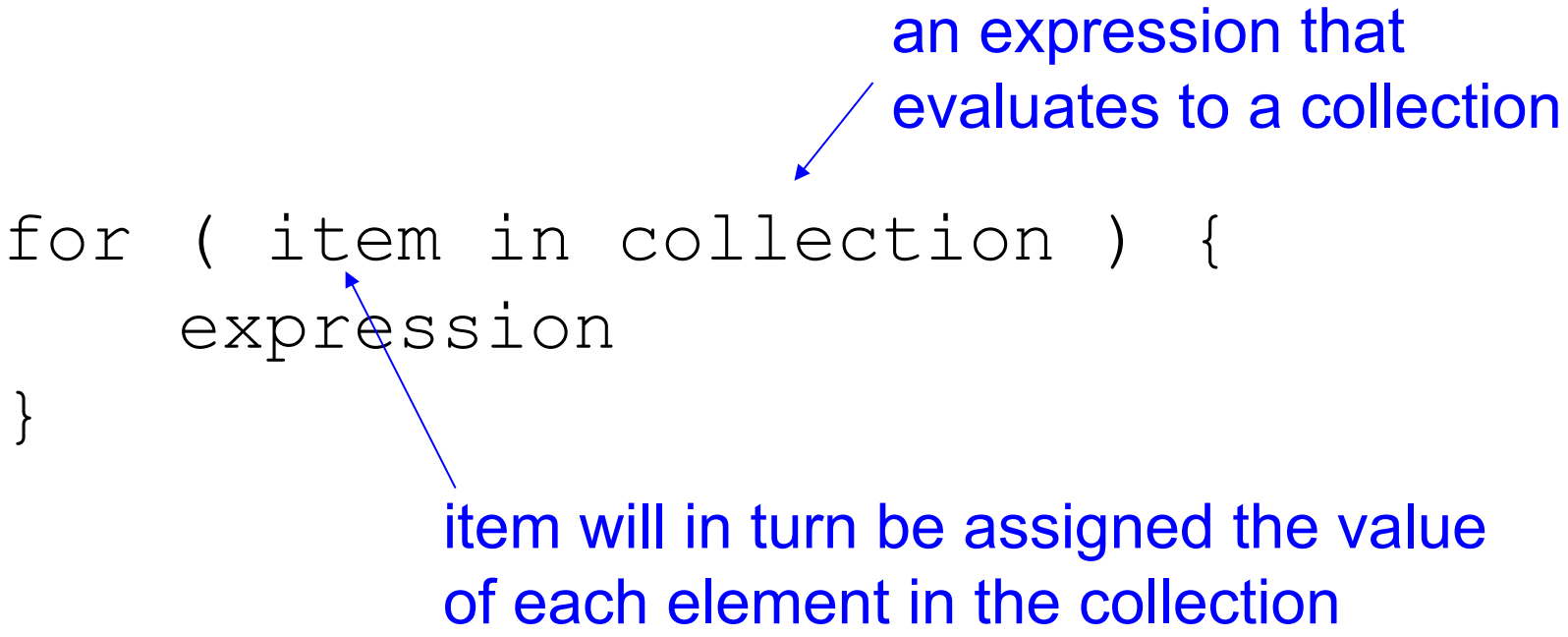
Collection controlled repetition

- Many languages have convenience structures for collection controlled repetition
- Often called foreach or similar
- General pseudocode:

```
for each item in collection
    do something
```

R: `for` is collection controlled

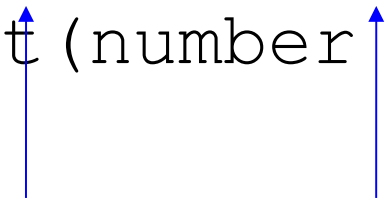
R's for structure is actually collection controlled repetition, a special case of counter controlled repetition



R: `for` is collection controlled

Example

```
a <- c(0.51,0.57,0.09,1.02,1.10)
for ( number in a ) {
    print(number * 2)
}
```



R: `for` is collection controlled

Example

```
a <- c(0.51,0.57,0.09,1.02,1.10)
for ( number in a ) {
    print(number * 2)
}
```

item collection

0.51
0.57
0.09
1.02
1.10

a

"number" is in turn each element of a

R: `for` is collection controlled

Example

```
a <- c(0.51,0.57,0.09,1.02,1.10)
for ( number in a ) {
    print(number * 2)
}
```

item collection

loop 1

a
0.51
0.57
0.09
1.02
1.10

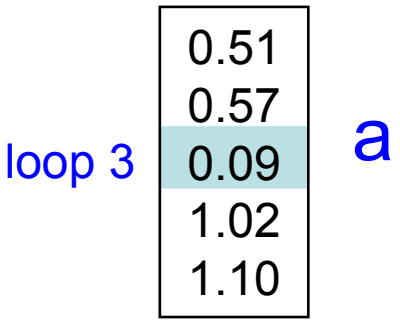
"number" is in turn each element of a

R: `for` is collection controlled

Example

```
a <- c(0.51,0.57,0.09,1.02,1.10)
for ( number in a ) {
    print(number * 2)
}
```

item collection



"number" is in turn each element of a

R: `for` is collection controlled

Example revisited

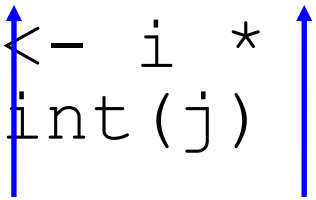
```
for ( i in 1:10 ) {
    j <- i * 2
    print(j)
}
```

R

R: `for` is collection controlled

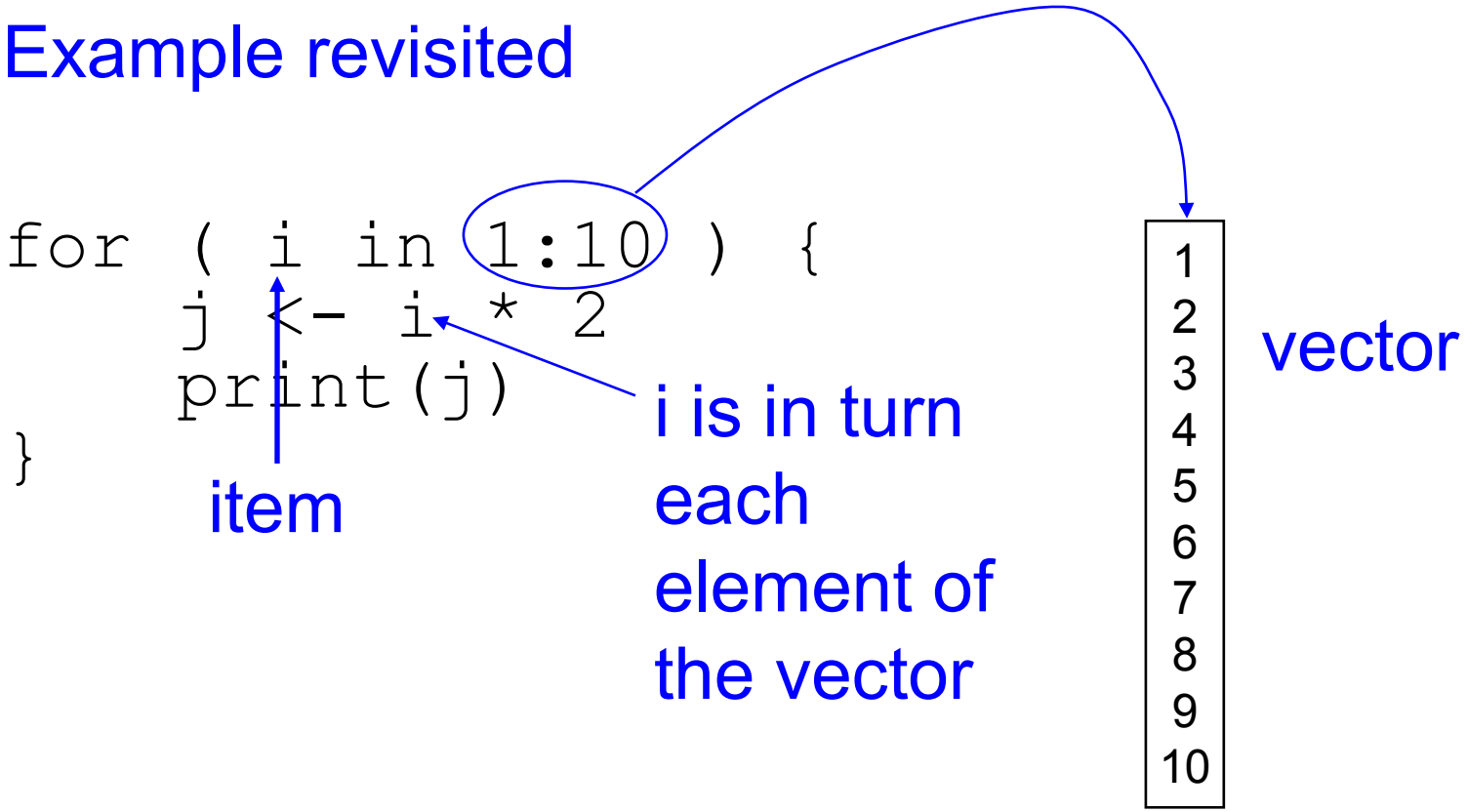
Example revisited

```
for ( i in 1:10 ) {
    j <- i * 2
    print(j)
}
```



R: `for` is collection controlled

Example revisited

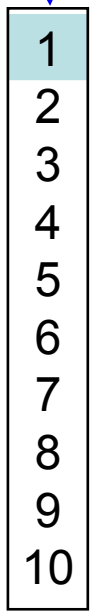


R: `for` is collection controlled

Example revisited

```
for ( i in 1:10 ) {
      j <- i * 2
      print(j)
}
```

i is in turn each element of the vector



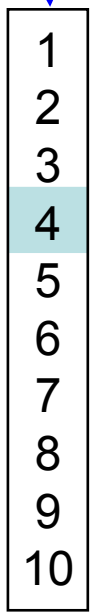
vector

R: `for` is collection controlled

Example revisited

```
for ( i in 1:10 ) {
      j <- i * 2
      print(j)
}
```

i is in turn each element of the vector



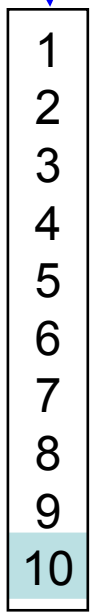
vector

R: `for` is collection controlled

Example revisited

```
for ( i in 1:10 ) {
      j <- i * 2
      print(j)
}
```

i is in turn each element of the vector



vector

Collection controlled repetition

Collection controlled repetition is a special case of counter controlled repetition

```
v <- ... #vector
n <- length(v)
i <- 1
while ( i <= n ) {
    expression using v[i]
    i <- i + 1
}
```

get vector length

index vector elements

optional

R

Collection controlled repetition

Collection controlled repetition is a special case of counter controlled repetition

```
v <- c(0.51,0.57,0.09,1.02,1.10)
n <- length(v)
i <- 1
while ( i <= n ) {
    j <- v[i] * 2
    print(j)
    i <- i + 1
}
```

Reduces to

```
for ( item in v ) {
    j <- item * 2
    print(j)
}
```

R

Data structure: list

- Collection of objects
- Can be heterogeneous

```r
mylist <- list(9, "A", 2)
```

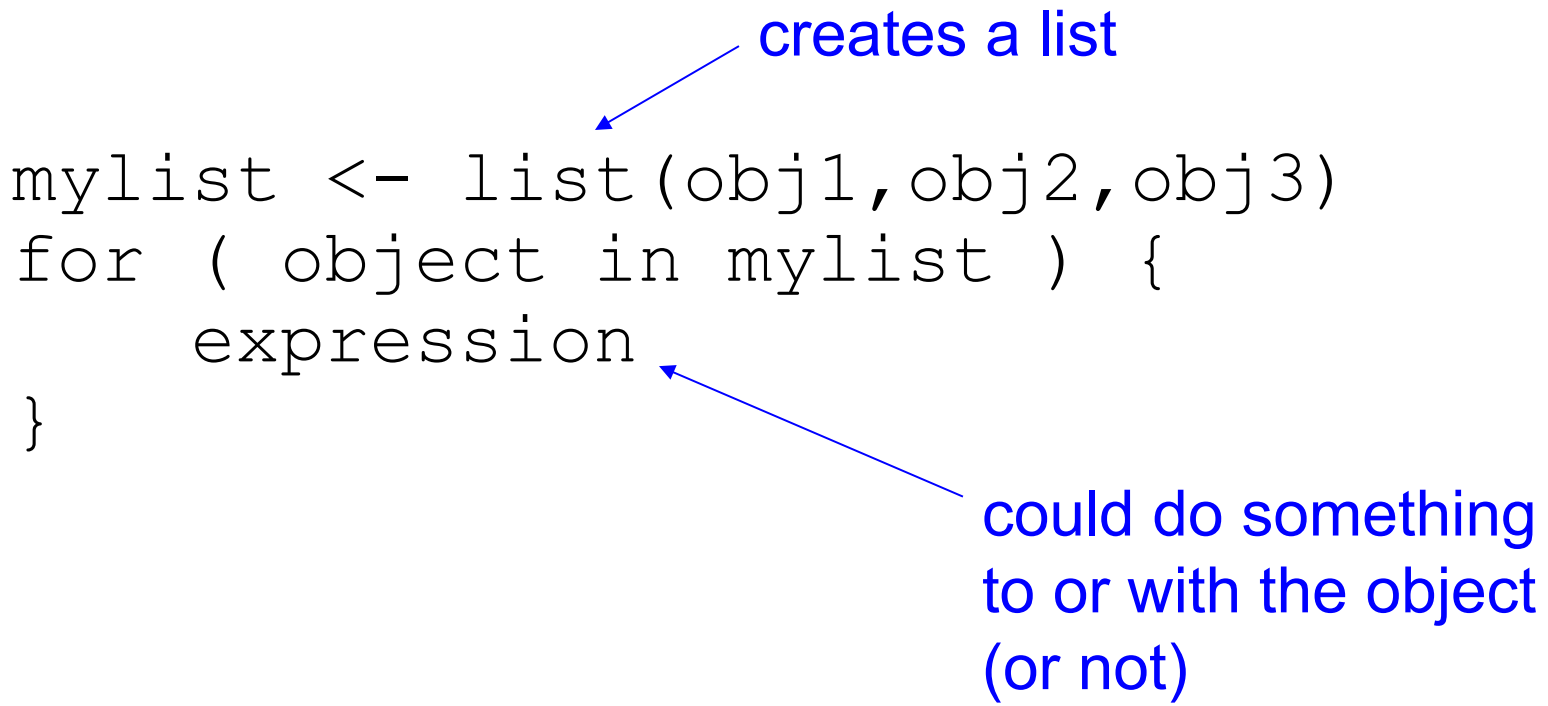
R

```python
mylist = ["a", 9, "b"]
```

Py

See additional code

R: collection control with lists



R: collection control with lists

Example

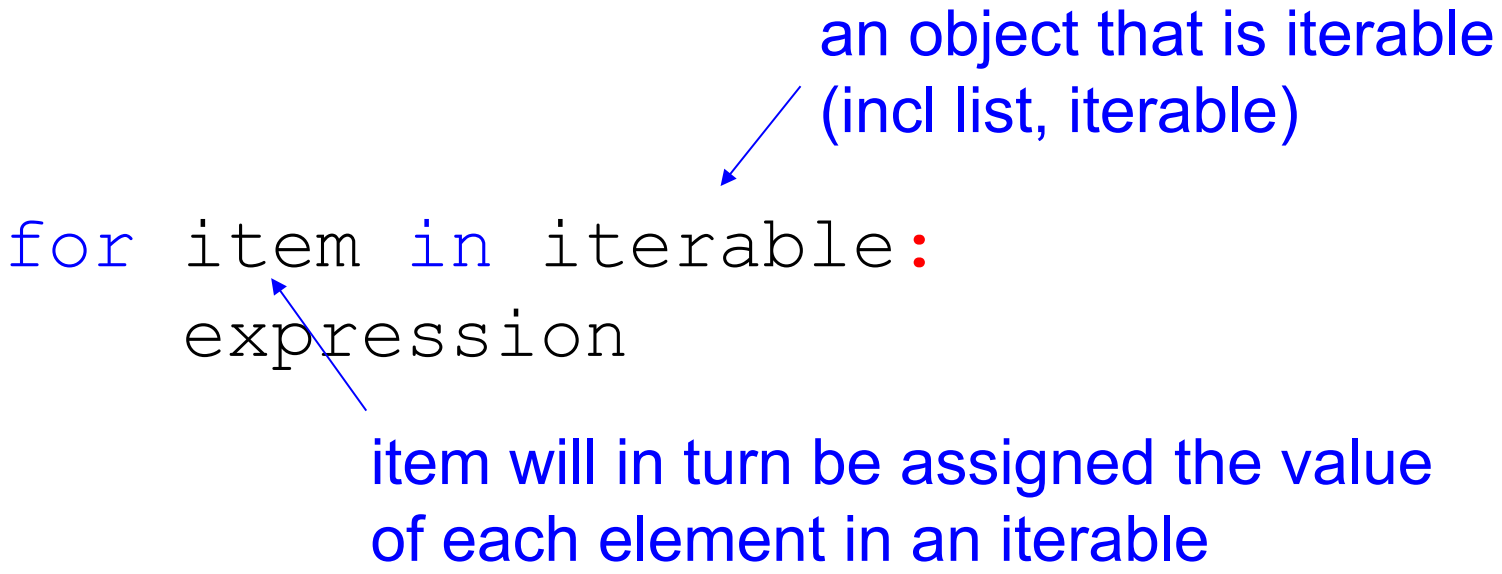
a, b, c, d
are numerical
vectors

```
datasets <- list(a, b, c, d)
for ( x in datasets ) {
    hist(x)
}
```

What does this do?

Python: `for` is iterator controlled

Often amounts to counter controlled repetition but sometimes sentinel controlled

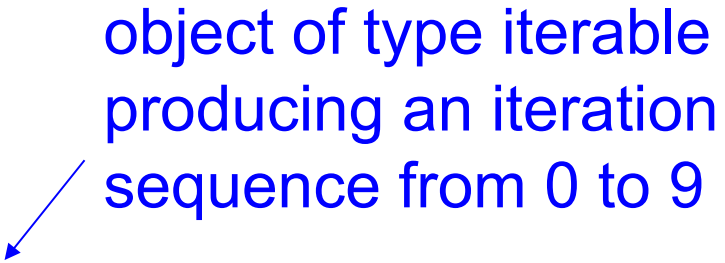


An iterator object has a start, an increment method, and checks for an end condition. Calculated iteratively.

Py

Python: `for`

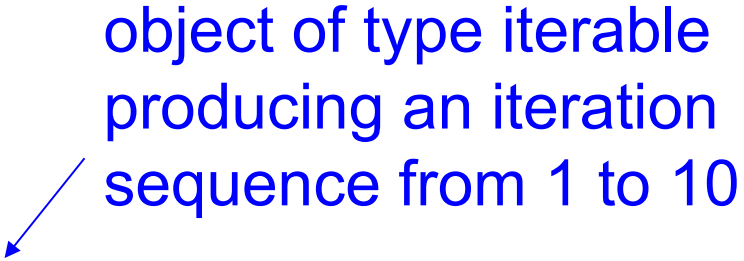
Classic counter controlled repetition




```
for i in range(10):
    print("Hello")
```

Python: `for`

General item in iterable




```
for i in range(1, 11):
    j = i * 2
    print(j)
```

Py

Python: `for`

Iterate over a collection

```
rainbow = list(range(1, 11))
for i in rainbow:
    j = i * 2
    print(j)
```

Py

Python: `for`

The list iterator checks for an end condition each iteration, which could change, so it's actually sentinel control

```
x = [0, 1]
for item in x:
    x.append(item)
```

lengthen x

This makes an infinite loop!

because the index of item never reaches the end

Py

Python: `for`

The list iterator is sentinel control

Effectively this:

```
x = [0, 1]
i = 0
while i < len(x):
    item = x[i]
    x.append(item)
    i = i + 1
```

never false

Contrast with counter control

```
x = [0, 1]
i = 0
n = len(x)
while i < n:
    item = x[i]
    x.append(item)
    i = i + 1
```

n calc ahead

Py

Python: `for`

Take away:

Mostly we can think of it as counter control even when it's implemented as sentinel control