

Programming Basics in Python

Building a Grading Machine

Yen-Yi Ho

Department of Statistics, University of South Carolina

2026-09-09

Today

- `if / elif / else`
- Vectorized conditionals: `np.where()`
- `for` loops
- Writing functions
- `map`, list comprehensions, and `.apply()`
- A simulation that ties it all together

Everything here is the Python counterpart of the R lecture. Watch for the **R vs. Python** boxes — that's where the traps are.

Setup

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats

np.set_printoptions(precision=4, suppress=True)
```

Section 1

Part I: if / else

The reciprocal example

R

```
a <- 0

if (a != 0) {
  print(1/a)
} else {
  print("No reciprocal for 0.")
}
```

Python

```
a = 0

if a != 0:
    print(1 / a)
else:
    print("No reciprocal.")
```

No reciprocal.

R vs. Python

R	Python
<code>if (cond) { } else { }</code>	<code>if cond: ... else:</code>
braces delimit the block	indentation (4 spaces) delimits it
a colon is not used	a colon ends the if line
<code>else if</code>	<code>elif</code>
<code>&&, , !</code>	<code>and, or, not</code>
<code>TRUE, FALSE</code>	<code>True, False</code>

***Careful.** Indentation is not cosmetic in Python — it is the syntax.*

The murders data

`dslabs::murders` is an R package; in Python we read the data with **pandas**.

```
url = ("https://raw.githubusercontent.com/rafalab/"
       "dslabs/master/inst/extdata/murders.csv")
try:
    murders = pd.read_csv("murders.csv")
except FileNotFoundError:
    murders = pd.read_csv(url)

murders.head(4)
```

	state	abb	region	population	total
0	Alabama	AL	South	4779736	135
1	Alaska	AK	West	710231	19
2	Arizona	AZ	West	6392017	232
3	Arkansas	AR	South	2915918	93

Murder rate per 100,000

```
# R: murder_rate <- murders$total / murders$population * 100000  
murder_rate = murders["total"] / murders["population"] * 100000  
murder_rate.head()
```

```
0      2.824424  
1      2.675186  
2      3.629527  
3      3.189390  
4      3.374138  
dtype: float64
```

which.min() becomes argmin()

Careful. The single biggest difference: R indexes from 1, Python from 0.

```
ind = murder_rate.argmin()      # 0-based position

print("position:", ind)
print("state:", murders["state"][ind])
print("rate:", round(murder_rate[ind], 4))
```

```
position: 45
state: Vermont
rate: 0.3196
```

Is any state below 0.5?

```
if murder_rate[ind] < 0.5:  
    print(murders["state"][ind])  
else:  
    print("No state has murder rate that low")
```

Vermont

Below 0.25?

```
if murder_rate[ind] < 0.25:  
    print(murders["state"][ind])  
else:  
    print("No state has a murder rate that low.")
```

No state has a murder rate that low.

Section 2

Part II: Build a Grading Machine

The scores

```
scores = [55, 82, 91, 40, 67, 78, 100, 62, 35, 88]  
scores
```

```
[55, 82, 91, 40, 67, 78, 100, 62, 35, 88]
```

Task. Write an if/else statement that reports **PASS** (score of 60 or more) or **FAIL** for the *second* student.

Pass / Fail

```
if scores[1] >= 60:          # scores[1] is 82 -- the 2nd student
    print("PASS")
else:
    print("FAIL")
```

PASS

Careful. The R version wrote `scores[2]`. In Python the second element is `scores[1]`.

Exercise 2: letter grades

```
score >= 90  ->  A
score >= 80  ->  B
score >= 70  ->  C
otherwise    ->  F
```

Do it for the **9th** student.

R

```
if (scores[9] >= 90) {  
  print("A")  
} else if (scores[9] >= 80) {  
  print("B")  
} else if (scores[9] >= 70) {  
  print("C")  
} else {  
  print("F")  
}
```

Python

```
if scores[8] >= 90:  
    print("A")          # scores[8] is 35  
elif scores[8] >= 80:  
    print("B")  
elif scores[8] >= 70:  
    print("C")  
else:  
    print("F")
```

F

Section 3

Part III: Vectorized Conditionals

ifelse() splits into two things

R's `ifelse()` does two different jobs. Python keeps them separate.

R	Python: one value	Python: whole vector
<code>ifelse(a > 0, 1/a, NA)</code>	<code>1/a if a > 0 else np.nan</code>	<code>np.where(a > 0, 1/a, np.nan)</code>

One value: the conditional expression

```
a = 0  
result = 1 / a if a > 0 else np.nan  
result
```

nan

A whole array: np.where()

```
a = np.array([0, 1, 2, -4, 5], dtype=float)

with np.errstate(divide="ignore"):  # silence the 1/0 warning
    result = np.where(a > 0, 1 / a, np.nan)

result

array([nan, 1. , 0.5, nan, 0.2])
```

Recoding a variable

```
b = np.array([0, 1, 1, 0])  
ans = np.where(b == 0, "MALE", "FEMALE")  
ans
```

```
array(['MALE', 'FEMALE', 'FEMALE', 'MALE'], dtype='<U6')
```

Missing values

```
R: no_nas <- ifelse(is.na(na_example), 0, na_example)
```

```
na_example = np.array([2.0, np.nan, 3.0, 1.0, np.nan,  
                        3.0, 2.0, np.nan, 5.0, 2.0])
```

```
print("missing before:", np.isnan(na_example).sum())
```

```
no_nas = np.where(np.isnan(na_example), 0, na_example)
```

```
print("missing after: ", np.isnan(no_nas).sum())
```

```
missing before: 3
```

```
missing after: 0
```

The pandas way

```
pd.Series(na_example).fillna(0).tolist()
```

```
[2.0, 0.0, 3.0, 1.0, 0.0, 3.0, 2.0, 0.0, 5.0, 2.0]
```

any() and all()

```
z = [True, True, False]
```

```
print(any(z))
```

```
print(all(z))
```

True

False

These two have the same names in both languages.

Section 4

Part IV: for Loops

Counting to five

R

```
for (i in 1:5) {  
  print(i)  
}
```

Python

```
for i in range(1, 6):  
    print(i)
```

1
2
3
4
5

Careful. *1:5 includes 5. range(1, 6) stops before 6.*

Filling a vector in a loop

```
m = 25
s_n = np.zeros(m)           # R: vector(length = m)

for n in range(1, m + 1):
    s_n[n - 1] = sum(range(1, n + 1))

s_n[:8]
```

```
array([ 1.,  3.,  6., 10., 15., 21., 28., 36.])
```

Skip the index arithmetic with enumerate()

```
s_n = np.zeros(m)

for i, n in enumerate(range(1, m + 1)):
    s_n[i] = np.arange(1, n + 1).sum()

s_n[:8]

array([ 1.,  3.,  6., 10., 15., 21., 28., 36.] )
```

Exercise: grade every student

```
scores = [55, 82, 91, 40, 67,  
          78, 100, 62, 35, 88]  
  
for i in scores:  
    if i >= 90:  
        print("A")  
    elif i >= 80:  
        print("B")  
    elif i >= 70:  
        print("C")  
    else:  
        print("F")
```

F
B
A
F
F
C
A
F
F
B

Section 5

Part V: Functions

function() becomes def

R

```
newfunction <- function(x, y){  
  ans <- x^2 + y^2  
  return(ans)  
}
```

Python

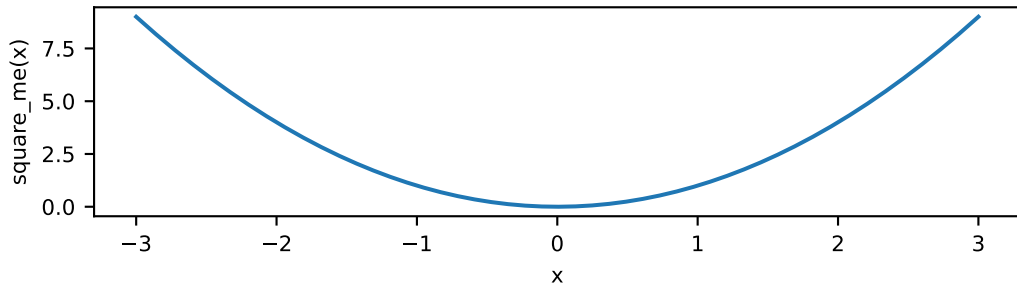
```
def newfunction(x, y):  
    ans = x**2 + y**2  
    return ans  
  
newfunction(3, 4)
```

25

Note. $\wedge$ in R is $**$ in Python. In Python $\wedge$ means bitwise XOR — a silent bug waiting to happen.

Plotting a function

```
def square_me(x):  
    return x**2  
  
x = np.arange(-3, 3.01, 0.01)  
plt.plot(x, square_me(x)); plt.xlabel("x"); plt.ylabel("square_me(x)")  
plt.tight_layout(); plt.show()
```



Writing our own average

```
def avg(x):  
    s = sum(x)  
    n = len(x)  
    return s / n      # R returns this implicitly -- Python will not  
  
x = list(range(1, 101))  
print(avg(x), np.mean(x))  
print(np.isclose(np.mean(x), avg(x)))
```

50.5 50.5

True

Careful. Without *return*, a Python function quietly hands back *None*.

Variables inside a function are local

```
s = 3

print(avg(range(1, 11)))    # the s inside avg() is a different s
print(s)                   # ... the global s is untouched
```

5.5

3

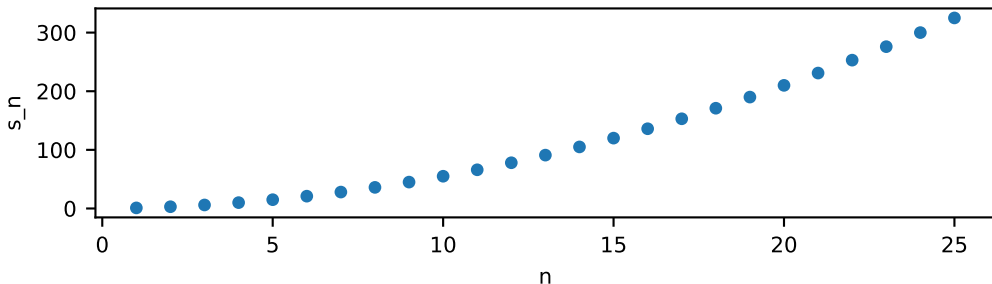
Default arguments

```
def avg(x, arithmetic=True):  
    x = np.asarray(x, dtype=float)  
    n = len(x)  
    return x.sum() / n if arithmetic else np.prod(x) ** (1 / n)  
  
print("arithmetic:", avg([1, 2, 3, 4]))  
print("geometric: ", avg([1, 2, 3, 4], arithmetic=False))
```

```
arithmetic: 2.5  
geometric: 2.213363839400643
```

compute_s_n

```
def compute_s_n(n):  
    return sum(range(1, n + 1))  
  
n = np.arange(1, m + 1)  
plt.scatter(n, [compute_s_n(i) for i in n], s=14)  
plt.xlabel("n"); plt.ylabel("s_n")  
plt.tight_layout(); plt.show()
```



Vectorization

```
x = np.arange(1, 11)
y = np.arange(1, 11)

print(np.sqrt(x))
print(x * y)
```

```
[1.      1.4142 1.7321 2.      2.2361 2.4495 2.6458 2.8284 3.
 3.1623]
[ 1  4  9 16 25 36 49 64 81 100]
```

But a list is not a vector!

```
print([1, 2, 3] * 2)           # repeats the list
print(np.array([1, 2, 3]) * 2) # doubles each element
```

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

[2 4 6]

Careful. *R has no such trap — every numeric vector is vectorized. In Python, use `np.array` when you want R-like behavior.*

Exercise: the grade_it function

```
def grade_it(grade):  
    if grade >= 90:  
        return "A"  
    elif grade >= 80:  
        return "B"  
    elif grade >= 70:  
        return "C"  
    else:  
        return "F"
```

```
grade_it(30)
```

'F'

Exercise: a loop that calls grade_it

```
letter_grade = ["NA"] * len(scores)    # R: rep("NA", length(scores))

for i in range(len(scores)):
    letter_grade[i] = grade_it(scores[i])

letter_grade
```

```
['F', 'B', 'A', 'F', 'F', 'C', 'A', 'F', 'F', 'B']
```

The more Pythonic version

```
letter_grade = []  
for score in scores:  
    letter_grade.append(grade_it(score))  
  
letter_grade
```

```
['F', 'B', 'A', 'F', 'F', 'C', 'A', 'F', 'F', 'B']
```

Section 6

Part VI: supply and Friends

Four ways to say `supply(x, sqrt)`

```
x = list(range(1, 6))

print(np.round([np.sqrt(i) for i in x], 3))    # 1. comprehension
print(np.round(list(map(np.sqrt, x)), 3))      # 2. map()
print(np.round(np.sqrt(x), 3))                # 3. vectorized ufunc
print(np.round(pd.Series(x).apply(np.sqrt).to_numpy(), 3)) # 4. apply
```

```
[1.      1.414 1.732 2.      2.236]
```

```
[1.      1.414 1.732 2.      2.236]
```

```
[1.      1.414 1.732 2.      2.236]
```

```
[1.      1.414 1.732 2.      2.236]
```

The **list comprehension** is the everyday choice. Reach for the **ufunc** when speed matters.

Exercise: the grading machine, vectorized

```
scores = [55, 82, 91, 40, 67, 78, 100, 62, 35, 88]
```

```
# R: sapply(scores, grade_it)
letter_grade = [grade_it(s) for s in scores]
letter_grade
```

```
['F', 'B', 'A', 'F', 'F', 'C', 'A', 'F', 'F', 'B']
```

The map() version

```
list(map(grade_it, scores))
```

```
['F', 'B', 'A', 'F', 'F', 'C', 'A', 'F', 'F', 'B']
```

A tidy gradebook

```
gradebook = pd.DataFrame({"score": scores})  
gradebook["letter"] = gradebook["score"].apply(grade_it)  
gradebook["pass"] = np.where(gradebook["score"] >= 60, "PASS", "FAIL")  
gradebook
```

	score	letter	pass
0	55	F	FAIL
1	82	B	PASS
2	91	A	PASS
3	40	F	FAIL
4	67	F	PASS
5	78	C	PASS
6	100	A	PASS
7	62	F	PASS
8	35	F	FAIL
9	88	B	PASS

Counting the grades

```
gradebook["letter"].value_counts().sort_index()
```

```
letter
```

```
A      2
```

```
B      2
```

```
C      1
```

```
F      5
```

```
Name: count, dtype: int64
```

Section 7

Part VII: apply() over a Matrix

Building the matrix

*Careful. R fills a matrix **column by column**. NumPy's reshape fills **row by row**. Use `order="F"` (Fortran order) to match R.*

```
# R: matrix(1:20, ncol=2)
mat = np.arange(1, 21).reshape(10, 2, order="F")
mat.T    # transposed here so it fits on the slide
```

```
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]])
```

Row and column means

```
cMean = mat.mean(axis=0)      # R: apply(mat, 2, mean)
rMean = mat.mean(axis=1)      # R: apply(mat, 1, mean)

print("cMean:", cMean)
print("rMean:", rMean)
```

```
cMean: [ 5.5 15.5]
rMean: [ 6.  7.  8.  9. 10. 11. 12. 13. 14. 15.]
```

The axis rule

What you want	R	NumPy
one number per column	<code>apply(m, 2, f)</code>	<code>m.f(axis=0)</code>
one number per row	<code>apply(m, 1, f)</code>	<code>m.f(axis=1)</code>

The numbering is the **mirror image** of R's. Read `axis=` as “*the axis that disappears*”: `axis=0` collapses the rows, leaving one value per column.

The literal translation

```
print(np.apply_along_axis(np.mean, 0, mat))  
print(np.apply_along_axis(np.mean, 1, mat))
```

```
[ 5.5 15.5]
```

```
[ 6.  7.  8.  9. 10. 11. 12. 13. 14. 15.]
```

The pandas version

```
df = pd.DataFrame(mat, columns=["V1", "V2"])
print(df.apply(np.mean, axis="index").tolist())
print(df.apply(np.mean, axis="columns").tolist())
```

[5.5, 15.5]

[6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0]

Section 8

Part VIII: Putting It Together

The simulation task

- 1 Generate 30 random numbers from a normal distribution
- 2 Randomly assign 15 to **control** and 15 to **treatment**
- 3 Run a t -test and report the p-value

Generating the data

```
rng = np.random.default_rng(123)          # R: set.seed(123)

obs = rng.normal(loc=0, scale=1, size=30)  # R: rnorm(30, 0, 1)
obs[:5]
```

```
array([-0.9891, -0.3678,  1.2879,  0.194 ,  0.9202])
```

```
rng.choice(["A", "B", "C"], size=1)        # R: sample(LETTERS[1:3], 1)
```

```
array(['B'], dtype='<U1')
```

Not good!

```
control <- sample(obs, 15)
trt      <- sample(obs, 15)
```

Two independent draws let the **same observation land in both groups**.

```
control_bad = rng.choice(obs, 15, replace=False)
trt_bad     = rng.choice(obs, 15, replace=False)

print("in BOTH groups:", len(np.intersect1d(control_bad, trt_bad)))
```

in BOTH groups: 8

This is good!

Sample the **indices** once; control gets the chosen ones, treatment the rest.

```
icontrol = rng.choice(30, size=15, replace=False) # R: sample(1:30, 15)

mask = np.zeros(30, dtype=bool)
mask[icontrol] = True

control = obs[mask]      # R: obs[icontrol]
trt = obs[~mask]         # R: obs[-icontrol]
```

Careful. R's `obs[-icontrol]` means “everything except”. In Python a negative index counts **from the end** — use a boolean mask and `~`.

The t-test

```
res = stats.ttest_ind(control, trt)

print("statistic:", round(res.statistic, 4))    # R: str(res)
print("p-value:  ", round(res.pvalue, 4))      # R: res$p.value
```

```
statistic: -0.822
p-value:    0.418
```

Do it 1000 times: the loop

```
sim, n = 1000, 15
rng = np.random.default_rng(1)

pvalue = np.zeros(sim)                # R: numeric(sim)

for i in range(sim):
    obs = rng.normal(size=2 * n)
    icontrol = rng.choice(2 * n, size=n, replace=False)
    mask = np.zeros(2 * n, dtype=bool)
    mask[icontrol] = True
    res = stats.ttest_ind(obs[mask], obs[~mask])
    pvalue[i] = res.pvalue

pvalue[:5]
```

```
array([0.2491, 0.0894, 0.9008, 0.2565, 0.0359])
```

Wrap it in a function

```
def getpvalue(i, n, rng):  
    obs = rng.normal(size=2 * n)  
    icontrol = rng.choice(2 * n, size=n, replace=False)  
    mask = np.zeros(2 * n, dtype=bool)  
    mask[icontrol] = True  
    res = stats.ttest_ind(obs[mask], obs[~mask])  
    return res.pvalue
```

Note. *i* is never used inside the function — it is just a placeholder so *map* (or *R*'s *sapply*) has something to iterate over. Don't name it *iter*: that is a Python built-in.

“supply” the simulation

```
# R: pvalue_sapply <- sapply(1:1000, getpvalue, n = 15)
rng = np.random.default_rng(1)
pvalue_lc = np.array([getpvalue(i, n=15, rng=rng)
                      for i in range(1000)])

rng = np.random.default_rng(1)
pvalue_map = np.array(list(map(lambda i: getpvalue(i, 15, rng),
                               range(1000)))))

np.array_equal(pvalue_lc, pvalue_map)    # R: identical(...)
```

True

Were we right?

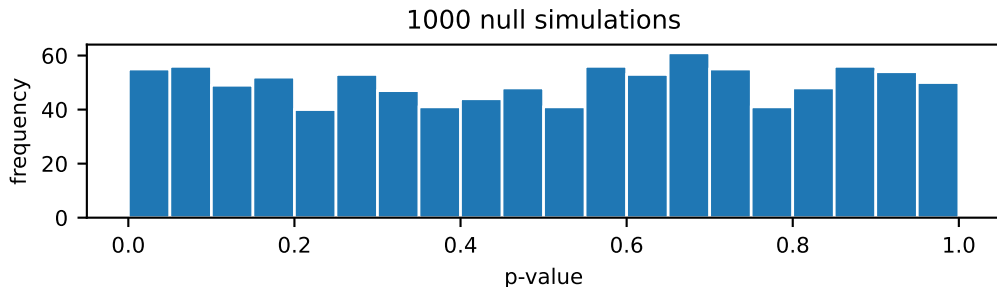
```
print("mean p-value:      ", round(pvalue_lc.mean(), 4))  
print("proportion < 0.05: ", (pvalue_lc < 0.05).mean())
```

```
mean p-value:      0.5039  
proportion < 0.05: 0.055
```

Under the null the p-value is **uniform on (0, 1)**, so its mean should be near 0.5 and about 5% of the draws should fall below 0.05.

The distribution of the p-values

```
plt.hist(pvalue_lc, bins=20, edgecolor="white")  
plt.xlabel("p-value"); plt.ylabel("frequency")  
plt.title("1000 null simulations")  
plt.tight_layout(); plt.show()
```



Section 9

Cheat Sheet

R to Python

R	Python
<code>if (x) { } else { }</code>	<code>if x: / else: (indentation, no braces)</code>
<code>else if</code>	<code>elif</code>
<code>ifelse(cond, a, b)</code>	<code>np.where(cond, a, b) / a if cond else b</code>
<code>is.na(x)</code>	<code>np.isnan(x) / pd.isna(x)</code>
<code>any(z) / all(z)</code>	<code>any(z) / all(z)</code>
<code>for (i in 1:5)</code>	<code>for i in range(1, 6):</code>
<code>length(x)</code>	<code>len(x)</code>
<code>x^2</code>	<code>x**2</code>
<code>function(x) { }</code>	<code>def f(x):</code>
implicit last value returned	explicit return required

R to Python, continued

R	Python
<code>sapply(x, f)</code>	<code>[f(i) for i in x] / list(map(f, x))</code>
<code>apply(m, 2, mean)</code>	<code>m.mean(axis=0)</code>
<code>apply(m, 1, mean)</code>	<code>m.mean(axis=1)</code>
<code>matrix(1:20, ncol=2)</code>	<code>np.arange(1,21).reshape(10,2, order="F")</code>
<code>rnorm(n)</code>	<code>rng.normal(size=n)</code>
<code>sample(1:n, k)</code>	<code>rng.choice(n, size=k, replace=False)</code>
<code>t.test(a,b)\$p.value</code>	<code>stats.ttest_ind(a,b).pvalue</code>
<code>set.seed(1)</code>	<code>rng = np.random.default_rng(1)</code>
<code>x[-i]</code> (drop i)	boolean mask <code>x[~mask]</code>
1-based indexing	0-based indexing

Questions?