recitation 01 (optional 6.86x notes)

You do not need to read these notes at all to get an A in this course; conversely, you may not cite these notes when solving homework or exams.

Recitation will have more coding practice and less detail than these notes. We'll skip whole passages of these notes during recitation. Footprints — roughly indicate our pacing: we will linger on the points near each footprint for three to seven minutes of recitation time.

A. prologue

bird's eye view

KINDS OF LEARNING — How do we communicate patterns of desired behavior? We can teach:
by instruction: "to tell whether a mushroom is poisonous, first look at its gills..."
by example: "here are six poisonous fungi; here, six safe ones. see a pattern?"
by reinforcement: "eat foraged mushrooms for a month; learn from getting sick."
Machine learning is the art of programming computers to learn from such sources. We'll focus on the most important case: learning from examples.°

← In Unit 5 of 6.86x, we'll see that learning by example is key to the other modes of learning.

FROM EXAMPLES TO PREDICTIONS — For us, a pattern of desired behavior is a function that for each given situation/prompt returns a favorable action/answer. Our goal is to write a program that, from a list of N examples of prompts and matching answers, determines an underlying pattern. We consider our program a success if this pattern accurately predicts answers corresponding to new, unseen prompts. We often define our program as a search, over some set $\mathcal{H}$ of candidate patterns, to minimize some notion of "discrepancy from the example data".

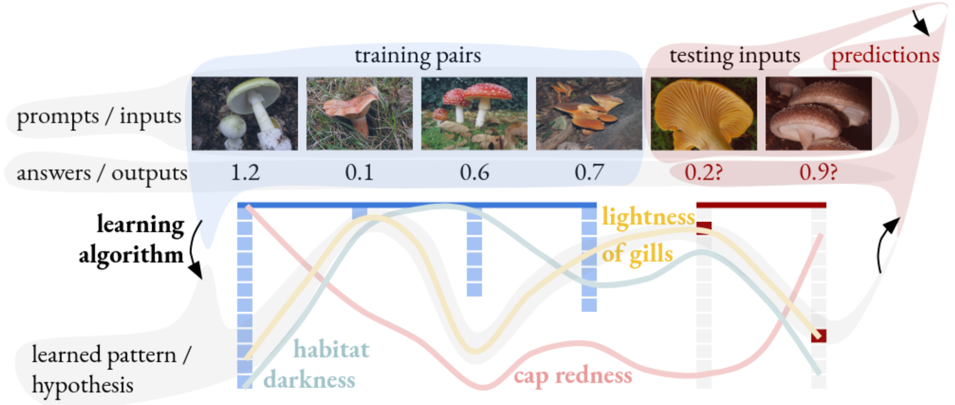

Figure 1: A program that learns to predict mushrooms' poison levels: first takes a list of labeled mushrooms as input (blue blob); searches through candidate patterns (here, the wiggly curves labeled `lightness of gills`, `habitat darkness`, and `cap redness`); and returns the pattern that best fits the examples. Evaluating this pattern on new mushrooms, we predict their poison levels (red blob).

The three bold black arrows show the flow of data from training examples to a learned pattern; and from that pattern together with testing prompts to predictions. Part of specifying the learning program is specifying the set of candidate patterns to consider.

To save ink, say that $\mathcal{X}$ is the set of possible prompts; $\mathcal{Y}$, of possible answers.° In the mushrooms example, $\mathcal{X}$ contains all conceivable mushrooms and $\mathcal{Y}$ contains all conceivable poison levels (perhaps all the non-negative real numbers).

← If we like, we can now summarize the data flow in symbols. A pattern is a function of type $\mathcal{X} \to \mathcal{Y}$. And we can model the examples from which our program learns as a list of type $(\mathcal{X} \times \mathcal{Y})^N$. Then a program that learns from examples has type:

$$\mathcal{L} : (\mathcal{X} \times \mathcal{Y})^N \to (\mathcal{X} \to \mathcal{Y})$$

Once we allow uncertainty by letting patterns map to *probability distributions* over answers, the type will change to:

$$\mathcal{L} : (\mathcal{X} \times \mathcal{Y})^N \to (\mathcal{X} \to \text{DistributionsOn}(\mathcal{Y}))$$

SUPERVISED LEARNING — We'll soon allow uncertainty by letting patterns map to *probability distributions* over answers. Even if the prompt is always the same — say, "produce a beautiful melody", we may seek to understand the complicated distribution over answers. We might regard our program a success if it can generate a variety of good answers. So-called **unsupervised learning** focuses in that way on output structure. By contrast, **supervised learning** (our subject in Unit 1), focuses on the input-output relation; it's interesting when the space of prompts is large.

a tiny example: handwritten digit classification

MEETING THE DATA — $\mathcal{X}$ = {grayscale 28×28-pixel images}; $\mathcal{Y} = \{1, 9\}$. Each datum (x, y) arises as follows: we randomly choose a digit $y \in \mathcal{Y}$, ask a human to write that digit in pen, and then photograph their writing to produce $x \in \mathcal{X}$.

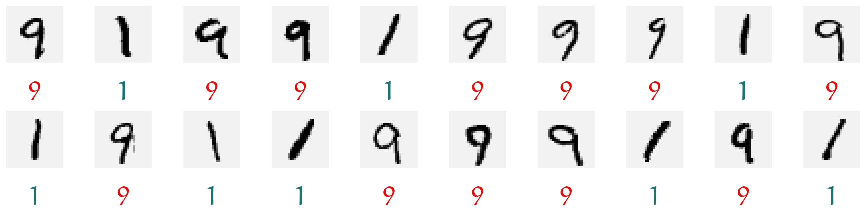

Figure 2: Twenty example pairs. Each photo x is a 28×28 grid of numbers representing pixel intensities. The light gray background has intensity 0.0; the blackest pixels, intensity 1.0. Below each photo x we display the corresponding label y: either $y = 1$ or $y = 9$. We'll adhere to this color code throughout this tiny example.

When we zoom in, we can see each photo's 28×28 grid of pixels. On the computer, this data is stored as a 28×28 grid of numbers: 0.0 for bright through 1.0 for dark. Our convention will be to name these 28×28 grid locations by the number of their row (counting starting from top) and then of their column (counting starting from the left). So location $(0, 0)$ is the upper left corner pixel; location $(27, 0)$ is the lower left corner pixel.

Exercise: Where is location $(0, 27)$? $(14, 8)$? $(14, 14)$?

Exercise: What pixel intensities do these two photos have at these three locations?

As part of getting to know the data, it's worth taking a moment to think about how we would go about hand-coding a digit classifier. The challenge is to complete the pseudocode "`if (?) then predict y=9 else predict y=1`". Well, 9s tend to have more ink than than 1s — should `(?)` threshold by the photo's darkness? Or: 1s and 9s tend to have different widths — should `(?)` threshold by the photo's dark part's width?

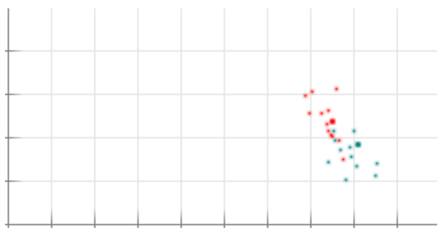

Figure 3: Our size-$N = 25$ set of training examples, viewed in the darkness-width plane. The vertical *darkness* axis ranges $[0.0, 0.25]$; the horizontal *width* axis ranges $[0.0, 0.5]$. The origin is at the lower left. Each cyan dot represents a $y = 1$ example; each red dot, a $y = 9$ one. The big 9 above has darkness and width $(0.118, 0.375)$; the big 1, $(0.092, 0.404)$. See where they are in this plot?

To make this precise, let's define a photo's *darkness* as its average pixel darkness; its *width* as the standard deviation of the column index of its dark pixels. For convenience let's normalize both width and darkness to have max possible value 1.0. Such functions from inputs in $\mathcal{X}$ to numbers are called **features**.

```
SIDE = 28
def darkness(x):
  return np.mean(np.mean(x))
def width(x):
  return np.std([col for col in range(SIDE)
                     for row in range(SIDE)
                     if 0.5 < x[row][col]  ])/(SIDE/2.0)
```

So we can threshold by darkness or by width. But this isn't very satisfying, since sometimes there are especially dark 1s or wide 9s. We thus arrive at the idea of using *both* features: 9s are darker than 1s *even relative to their width*. So we might write something like `2*darkness(x)-width(x)>0` for° our condition.

← That factor of 2 comes from our observation that darkness and tends to be 0.15 while width tends to be around 0.30.

```
def hand_coded_predict(x):
  return 9 if 2*darkness(x)-width(x)>0 else 1
```

Exercise: Beyond width and darkness, what features do you think might help us to separate digits 1 from 9? How about 0 from 9?

CANDIDATE PATTERNS — We can generalize the hand-coded hypothesis from the previous passage to other coefficients besides $1 \cdot \text{width}(x) - 2 \cdot \text{darkness}(x)$. We let our set $\mathcal{H}$ of candidate patterns contain all “linear hypotheses” $f_{a,b}$ defined by:

$$f_{a,b}(x) = 9 \;\; \text{if} \;\; a \cdot \text{darkness}(x) + b \cdot \text{width}(x) > 0 \;\; \text{else} \;\; 1$$

Each $f_{a,b}$ makes predictions of ys given xs. As we change a and b, we get different predictors, some more accurate than others.

```
def predict(x,a,b):
  return 9 if a*width(x) + b*darkness(x) > 0 else 1
```

Exercise: how do ■□□□’s 3 straight lines and □■□□’s 3 marked points correspond?

OPTIMIZATION — Let’s write a program $\mathcal{L}$ that given a list of *training examples* produces a hypothesis in $h \in \mathcal{H}$ that helps us predict the labels y of yet-unseen photos x (*testing examples*). Insofar as training data is representative of testing data, it’s sensible to return a $h \in \mathcal{H}$ that correctly classifies maximally many training examples. To do this, let’s make $\mathcal{L}$ loop over all integer pairs (a, b) in $[-99, +99]$:

```
def accuracy_on(examples,a,b):
  return sum(1.0 for x,y in examples if predict(x,a,b)==y)/len(examples)

def best_hypothesis():
  return max((accuracy_on(training_data, a, b), (a,b))
             for a in range(-99,100)
             for b in range(-99,100))
```

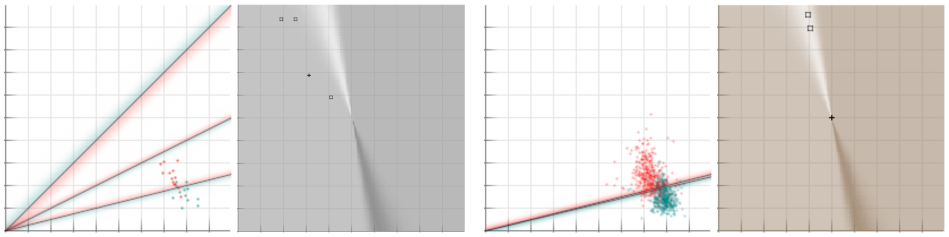
Figure 4: **Training (■■□□) and testing (□□■■).** 3 hypotheses classify training data in the darkness-width plane (■□□□). Each point in the (a, b) plane (□■□□) represents a hypothesis; we shade them by the fraction of training data they misclassify: darker means less accurate. Panes □□■■ show the same for *testing* data. ■□■□’s axes range $[0, 0.5]$. □■□■’s axes range $[-99, +99]$.

When we feed $N = 25$ training examples to $\mathcal{L}$, it produces $(a, b) = (80, -20)$ as a minimizer of **training error**, i.e., of the fraction of training examples misclassified. It misclassifies only 12% of training examples. Yet the same hypothesis misclassified a greater fraction — 18% — of fresh, yet-unseen testing examples. That latter number — called the **testing error** — represents our program’s accuracy “in the wild”; it’s the number we most care about. The difference between training and testing error is the difference between our score on a practice exam or homework, where we’re allowed to review mistakes we made and do a second try, versus our score on an exam, where we don’t know the questions beforehand and aren’t allowed to change our answers once we get our grades back.
Exercise: visualize $f_{a,b}$’s error on $N = 1$ example as a function of (a, b).

ERROR ANALYSIS — Intuitively, our testing error of 18% comes from three sources: **(a)** the failure of our training set to be representative of our testing set; **(b)** the failure of our program to exactly minimize training error over $\mathcal{H}$; **(c)** and the failure of our hypothesis set $\mathcal{H}$ to contain “the true” pattern.

These errors are respectively called **generalization error**, **optimization error**, **approximation error**.

Here, we got optimization error $\approx 0\%$ (albeit by *unscalable brute-force*). Because optimization error is zero in our case, the approximation error and training error are the same: $\approx 12\%$. The approximation error is so high because our straight lines are *too simple*: width and darkness lose useful information, and the "true" boundary between training digits looks curved. Finally, our testing error $\approx 18\%$ exceeds our training error. We thus suffer a generalization error of $\approx 6\%$: we *didn't perfectly extrapolate* from training to testing situations. In 6.86x we'll address all three italicized issues.

Exercise: why is generalization error usually positive?

supervised learning framework

FORMALISM — Draw training examples $\mathcal{S} : (\mathcal{X} \times \mathcal{Y})^N$ from nature's distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$. A pattern $f : \mathcal{X} \to \mathcal{Y}$ has **training error** $\text{trn}_{\mathcal{S}}(f) = \mathbb{P}_{(x,y)\sim\mathcal{S}}[f(x) \neq y]$, an average over examples; and **testing error** $\text{tst}(f) = \mathbb{P}_{(x,y)\sim\mathcal{D}}[f(x) \neq y]$, an average over nature. A *learning program* is a function $\mathcal{L} : (\mathcal{X} \times \mathcal{Y})^N \to (\mathcal{X} \to \mathcal{Y})$; we want to design $\mathcal{L}$ so that it maps typical $\mathcal{S}$s to fs with low $\text{tst}(f)$.

LEARNING ERROR — As in the previous subsection's tiny example, we often define $\mathcal{L}$ to roughly minimize $\text{trn}_{\mathcal{S}}$ over a set $\mathcal{H} \subseteq (\mathcal{X} \to \mathcal{Y})$ of candidate patterns. Then tst decomposes into the failures of $\text{trn}_{\mathcal{S}}$ to estimate tst (generalization), of $\mathcal{L}$ to minimize $\text{trn}_{\mathcal{S}}$ (optimization), and of $\mathcal{H}$ to contain nature's truth (approximation):

$$\begin{aligned}
\text{tst}(\mathcal{L}(\mathcal{S})) = \; & \text{tst}(\mathcal{L}(\mathcal{S})) && - \text{trn}_{\mathcal{S}}(\mathcal{L}(\mathcal{S})) && \}\ \textbf{generalization}\text{ error} \\
& + \text{trn}_{\mathcal{S}}(\mathcal{L}(\mathcal{S})) && - \inf_{\mathcal{H}}(\text{trn}_{\mathcal{S}}(f)) && \}\ \textbf{optimization}\text{ error} \\
& + \inf_{\mathcal{H}}(\text{trn}_{\mathcal{S}}(f)) && && \}\ \textbf{approximation}\text{ error}
\end{aligned}$$

These terms are in tension. For example, as $\mathcal{H}$ grows, the approx. error may decrease while the gen. error may increase — this is the "**bias-variance** tradeoff".

WORKFLOW — *Framing* is about knowing our data and knowing what problem we want it to help solve. Framing includes *factorizing* a complex prediction problem into simple classification and regression problems; preparing and *splitting* the consequent example pairs into training, development, and testing sets, and and *visualizing* the training data to weigh our intuitions.

Modeling is about presenting our data in a way that is easy to digest for the computer — in a way that easily found patterns are intuitively likely to be accurate. This is the key stage where we will inject our *domain knowledge* — that is, our human experience and intuition about which factors are likely to help with prediction. Modeling includes *featurizing* our inputs and *regularizing*

by choosing a class of *priors* and symmetries.

Training is about getting the computer to digest the data we've given it. Under the hood, the computer will try a zillion candidate patterns until it hits upon one that explains the examples relatively well. Above, we used brute force search; soon we'll learn more efficient approximate algorithms such as *gradient descent*.

Training also involves *model selection*: selecting from several qualitatively different models one that best predicts the development set.

Harvesting is about doing stuff with the pattern the computer settled on after it tried a zillion candidates. We can derive insights from the pattern itself (which factors ended up being most important?) and we can predict outputs corresponding to fresh inputs. Qualifying both applications is the pattern's quality. To assess this, we measure its accuracy on our held-out testing data.

appendix: python programming

python setup

Displace one note and there would be diminishment,
displace one phrase and the structure would fall.
— antonio salieri, on wolfgang mozart's music, as untruthfully portrayed in *Amadeus*

WHAT'S PYTHON? — Python is a popular programming language. Its heart is the **Python interpreter**, a computer program that takes a plain old text file such as this two-liner° —

```
print('hello,␣world!')
print('cows␣and␣dogs␣are␣fluffy')
```

← The instruction `print` just displays some text onto our screen. For example, the first line of this program displays `hello, world!` onto our screen. This instruction doesn't rely on or activate any ink-on-paper printing machines.

— and executes the instructions contained in that text file. The jargon is that these textual instructions are **source code**.

The instructions have to be in a certain, extremely rigid format in order for the interpreter to understand and execute them. That's why they call Python a *language*: it has its own rigid grammar and its own limited vocabulary. If the interpreter encounters incorrectly formatted instructions — if even a single punctuation mark is missing or a — the interpreter will display a bit of text in addition to the word `Error` and immediately after abort its mission.°

← Adventure boldly when learning Python! It might feel catastrophic when you encounter an error and the interpreter 'dies'. But (unless you go out of your way to use special instructions we won't teach in class) these errors won't hurt your computer. There aren't any lasting effects. So **errors are hints, not penalties**. If you encounter an error, just modify your instructions to address that error, then run the interpreter again. Engage in a fast feedback cycle (*I'll try this... error... okay how about this... different error... hmm let's think...*) to learn to program well.

We'll use Python in 6.86x to instruct our computer to analyze and learn from data. The gigantic project of instructing a computer to learn is a bit like teaching a person by mail. We never see them: we only exchange words back and forth. They have never seen a horse, and yet we want to teach them to distinguish horses from zebras from donkeys from tapirs from rhinos. Though severely limited in their vocabulary and their ability to appreciate analogies and similarities, they are extraordinarily meticulous, patient, and efficient. That's what programming will be like.

At this point, you might have several questions:

Picking up the pen: How do I install and use the Python interpreter?

Writing sentences: What syntax rules must my instructions obey in order for the interpreter to understand that I want it do such-and-such task?

Composing essays: How do I organize large programs?

Teaching via mail: What instructions make make the computer learn from data?

This subsection and the next three subsections address these four questions in turn.

WHICH THINGS WE'LL SET UP — Let's set up Python by installing the Python interpreter. Actually, I should say *a* Python interpreter: each of the many software tools we'll use in 6.86x has a zillion versions; it can get confusing tracking which versions coexist and which clash. We will use **Python version 3.8** throughout 6.86x.

Beyond the Python interpreter, there is an ecosystem of useful tools: machine learning modules that help us avoid reinventing the wheel; a rainbow of feature-rich text editors specialized for writing Python code; and a package manager called `conda` that eases the logistics nightmare of coordinating the versions we have of each tool.

SETUP ON WINDOWS —

SETUP ON MACOS —

SETUP ON LINUX — I'll assume we're working with Ubuntu 20 or related. If you're on a different Linux, then similar steps should work — google search to figure out how.

FILL IN: how to run, comments, basic stdIO line by line HELLO WORLD; string type line by line FAHRENHEIT KELVIN CELSIUS; float and integer types

python basics

If I have not seen as far as others, it is because giants were standing on my shoulders.
— hal abelson

Once you learn how to program, I highly recommend reading Edsger Dijkstra's *A Discipline of Programming*. That book contains many beautiful viewpoints that made me a better programmer.

OUTPUT AND SEQUENCING — Try this one-liner:

```
print('hello!')
```

Now try this:

```
print(686)
```

We execute multiple instructions in sequence by placing them on successive lines:

```
print('hello!')
print(6)
print(86)
print('goodbye!')
```

STATE AND ASSIGNMENT — Try this two-liner:

```
x = 686
print(x)
```

And this three-liner:

```
x = 686
x = 123
print(x)
```

Does the following print the same thing?

```
x = 123
x = 686
print(x)
```

What do you predict this does:

```
x = 123
print(x)
x = 686
print(x)
```

Above, `x` names a **variable**. A variable stores some value at each point in time; it might store different values at different times. We make a variable just by mentioning it in an **assignment** instruction such as `x=123`. We get to choose each variable's name however we'd like; a variable may have a one-letter name or a longer name:◦

← Naming rules: the first character must belong to the English alphabet (so there are 26 lower case choices plus 26 upper case choices). Each subsequent character is either an underscore (that's the symbol that looks like _), a digit (`0` through `9`), or a member of the English alphabet. The following names are forbidden, since they already mean something else in Python: `assert`, `bool`, `break`, `chr`, `class`, `continue`, `def`, `dict`, `elif`, `else`, `enumerate`, `except`, `float`, `for`, `if`, `import`, `input`, `int`, `lambda`, `len` `list`, `max`, `min`, `ord`, `print`, `range`, `round`, `set`, `str`, `sum`, `try`, `type`, `while`, `with`, `zip`. For a full list of forbidden names, google search *list of Python keywords and built-in functions*.

```
my_favorite_number = 123
print(my_favorite_number)
my_favorite_number = 686
print(my_favorite_number)
```

We can use multiple **variables**:

```
x = 123
y = 686
print(x)
```

What do you think the following prints?

```
x = 123
y = 686
x = y
print(x)
print(y)
```

ARITHMETIC — We can put math on the right hand side of assignments:

variable cube; assignment as shearing

```
x = 123
y = x*2 + 1000
print(x)
print(y)
```

What do you think the following prints?

```
x = 123
x = x*2 + 1000
print(x)
```

What do you think the following prints?

```
x = 123
x = x*2 + 1000
print(x)
```

Here's a more involved example. See what it's trying to do?

```
target = 20.0
x = 4.0
x = x - 0.01*(x*x*(x - target/x))
x = x - 0.01*(x*x*(x - target/x))
x = x - 0.01*(x*x*(x - target/x))
x = x - 0.01*(x*x*(x - target/x))
x = x - 0.01*(x*x*(x - target/x))
x = x - 0.01*(x*x*(x - target/x))
y = x*x
message = str(x) + '␣squares␣to␣' + str(y) + '␣which␣nearly␣equals␣' + str(target)
print(message)
```

If you successfully executed the above code, then you have officially done **iterative optimization**! Notice the repeated instruction `x=(target/x+x)/2`. Each time this instruction is executed, `x`'s value gets closer to the square root of `target`'s value. 6.86x features this theme of "repeatedly improve in order to optimize" in the guises of **gradient descent** and of **expectation maximization**. In fact, the code above is gradient descent on the loss function $\ell(x) = (x^2 - \text{target})^2$.

INTS, FLOATS, STRINGS, BOOLS — The variable values we work with might be numbers, or pieces of text, or answers to a true-false question, or something else. The kind of thing a value is is called that value's **type**. We use the jargon **string** or **str** for the type of thing that chunks of text are. For example, much as `3.141` is a number, `'hello, world!'` is a string. The jargon **bool** denotes the type of thing that can answer a true-or-false question. For example, while the value `'True'` is a string, the value `True` is a bool. So is `False`. Those are the only two possible bools. We'll distinguish two kinds of numeric value: values who by dint of their type are integers, and values that might or might not be integers. The jargon word for the former's type is **int**; for the latter's, **float**.° For example, when we write `5` or `686`, the interpreter understands those values to be ints; when we write `5.0` or `0.003141`, the interpreter understands those values to be floats.

← Historically, **int** abbreviates *integer* and **float** stands for *floating decimal point* (except with binary instead of decimal). The poetic image is of a decimal point that can drift around as if in a sea: `31.41`, `3.141`, `0.3141`, `0.03141`, etc. Floats thus stand in contrast to "banker's" numbers, which do not track differences smaller than one-hundredth of a dollar.

Different types support different operations and activities. For example, it makes sense to multiply two floats but not two strings. And if `x` is a variable containing a value of type `int`, then it makes sense to access a string's `x`th character. But that wouldn't make sense with `int` replaced by `float`.

Here's a flash summary of several common operations:

OPERATION	INPUT TYPE(S)	OUTPUT TYPE	HOW TO WRITE IT
add	`int` , `int`	`int`	`x+y`
add	`int` , `float`	`float`	`x+y`
add	`float` , `int`	`float`	`x+y`
add	`float` , `float`	`float`	`x+y`
literal int 5		`int`	`5`
literal float 5.0		`float`	`5.0`
count characters	`str`	`int`	`len(s)`
nth character	`str` , `int`	`str`	`s[n]`
concatenate	`str` , `str`	`str`	`left_str+right_str`
literal string '5'		`str`	`'5'`
round toward zero	`float`	`int`	`int(f)`
promote to float	`int`	`float`	`float(n)`
render	`int`	`str`	`str(n)`
render	`float`	`str`	`str(f)`
parse	`str`	`int`	`int(s)`
parse	`str`	`float`	`float(s)`
negation	`bool`	`bool`	`not x`
conjunction	`bool` , `bool`	`bool`	`x and y`
disjunction	`bool` , `bool`	`bool`	`x or y`
literal bool False		`bool`	`False`
check equality	`str` , `str`	`bool`	`x==y`
check equality	`int` , `int`	`bool`	`x==y`
check at-most	`int` , `int`	`bool`	`x<=y`
check at-least	`int` , `int`	`bool`	`x>=y`
check at-most	`float` , `float`	`bool`	`x<=y`
check at-least	`float` , `float`	`bool`	`x>=y`

Table 1: Python supports all of addition, subtraction, multiplication, and exponentiation: we use the symbols `+`, `-`, `*`, `/`, and `**`. For brevity we listed only addition in this table.

That's a lot to take in. Let's digest by seeing these operations in action:

```
my_string = 'dear␣world,'
my_string = my_string + '␣my␣favorite␣number␣is␣' + str(389)
print(my_string)
```

And

```
s = 'hi,␣world!'
my_message = 'my␣string␣s␣has␣' + str(len(my_string)) + 'many␣characters,'
my_message = my_message + '␣of␣which␣the␣first␣and␣last␣couple␣are:'
my_message = my_message + s[0] + s[1] + '...' + s[8] + s[9]
print(my_message)
```

And

```
x = 3 * 5.01
y = 3.01 * 5
s = 'abcdefghijklmnopqrstuvwxyz'
is_nearly_equal = (y-0.1 < x) and (x < y+0.1)
print(is_nearly_equal)
```

INPUT —

CONDITIONALS —

ITERATION —

NESTING, SCOPE, INDENTATION — skip, abort

REMARKS ON 'TYPES' —

DATA IN AGGREGATE — lists, strings, numpy arrays dictionaries comprehensions functional idioms

EXAMPLE: THE RESTAURANT BILL — Let's write a program that takes a list of prices as command line input and displays a grand total including with 6% tax and a variety of tips: 9%, 12%, 15%, 18%, 21%. We'll write the program in two styles. Here is a more conceptual style:

```
import sys
prices = [float(arg) for arg in sys.argv[1:]]
total = sum(prices)

with_tax = total * (1.0 + 0.06)
print('raw␣price:', total   , 'dollars')
print('with␣tax:' , with_tax, 'dollars')
for tip in range(9, 24, 3):
  with_tax_and_tip = total * (1.0 + 0.06 + tip/100.0)
  print('tip␣=', tip, 'percent', '-->', with_tax_and_tip, 'dollars')
```

We can say the same thing more explicitly as follows:

```
import sys
prices = []
for arg in sys.argv[1:]:
  prices.append(float(arg))
total = 0.0
for price in prices:
  total = total + prices

print('raw␣price:', total                , 'dollars')
print('with␣tax:' , total * (1.0 + 0.06), 'dollars')
for tip in range(9, 24, 3):
  print('tip␣=', tip, 'percent', '-->', total * (1.0 + 0.06 + tip/100.0), 'dollars')
```

EXAMPLE: EQUATION SOLVER — Let's write a program that takes a one-variable math equation as a command line argument and displays a solution. We'll restrict ourselves to polynomial equations such as `xxxxx=xx+xx+1+1` (our way of writing $x^5 = 2x^2 + 2$). That is, we'll assume the equation has no spaces and consists of only the characters `x`, `=`, `+`, `1`, `0`. Two more valid equations are `xxx+xx+x+1=0` and `x=xx`.

```
# step 0: read the equation-to-solve from the command line
import sys
import numpy as np
equation = sys.argv[1]

# step 1: check whether the equation holds for each x in a large
#         range with fine spacing; print and exit if for one of these
#         x's the equation holds to tolerance plus-minus 0.1
for x in np.arange(-100.00, +100.01, 0.01):

  # step 1a: initialize accumulator variables that will eventually
  #          represent the values of the equations's left hand and
  #          right hand sides
  index = 0
  left_hand_side = 0.0
  right_hand_side = 0.0
  term = 1.0

  # step 1b: compute the left hand side's numeric value;
  #          the left hand side ends the moment we see an equals sign
  while equation[index] != '=':
    if   equation[index] == '0': term = term * 0.0
    elif equation[index] == '1': term = term * 1.0
    elif equation[index] == 'x': term = term * x
    else:
      left_hand_side = left_hand_side + term
      term = 1.0

  # step 1c: skip the equals sign
  index = index + 1

  # step 1d: compute the right hand side's numeric value;
  #          the right hand side ends when our equation ends
  while index != len(equation):
    if   equation[index] == '0': term = term * 0.0
    elif equation[index] == '1': term = term * 1.0
    elif equation[index] == 'x': term = term * x
    else:
      right_hand_side = right_hand_side + term
      term = 1.0

  # step 1e: check whether the left and right hand sides are nearly
  #          equal.  if they are, report and exit
  difference = left_hand_side-right_hand_sided
  if -0.1<difference and difference<+0.1:
    print('x␣=', x, 'solves', equation, 'with␣error␣only', difference)
    sys.exit()

# step 2: report that the search for a solution was unsuccessful.
#         since step 1e exits when a solution is found, the
#         interpreter will get to step 2 only if no solution is found
print('no␣solution␣found␣to␣desired␣tolerance')
```

structuring code

ROUTINES — `defd` functions (ordinary args, return) interfaces: (kwargs; None as default return value; sentinels; more) code architecture and hygiene

MORE INPUT/OUTPUT — print formatting and flushing basic string manipulations (strip, join, etc) file io command line arguments example: read csv random generation and seeds??

HOW NOT TO INVENT THE WHEEL — matplotlib.plot numpy os package managers etc

numpy and friends

ARRAYS AKA TENSORS —

IMAGES: VIEWING, REPRESENTING, RENDERING —

GRADIENT DESCENT DONE MANUALLY —

GRADIENT DESCENT IN PYTORCH —

GRADIENT DESCENT IN TENSORFLOW —