OReilly - 2024-02February-08

February 22, 2024

1 Agenda

1. Fundamentals and core concepts
    - Syntax of Python
    - Values and variables
    - Printing things on the screen
    - Getting input from the user
    - Making decisions with `if` and `else`
    - Numbers
    - Strings (aka text)
    - Methods – what are they?
2. Loops, lists, and tuples
    - What are loops (`for` and `while`)
    - Looping over strings
    - Looping over numbers
    - Lists
        - Storing in lists
        - Retrieving from lists
    - Turning strings into lists, and then lists into strings
    - Tuples and tuple unpacking
3. Dictionaries and files
    - What are dicts?
    - Reading from them, writing to them
    - The different paradigms for working with dicts
    - How do they work behind the scenes?
    - Reading from and writing to text files
4. Functions
    - What is a function?
    - Writing functions
    - Arguments and parameters
    - Local vs. global variables
    - Return values
5. Modules and packages
    - What are these, and why do we need them?
    - Using modules that Python provides
    - Writing our own modules
    - Third-party packages and downloading them from PyPI
    - What next?

2 What is Jupyter?

Jupyter is a Web-based interface for Python. If you're using Jupyter, then you have the illusion of writing Python inside of your browser.

You can install Jupyter on your own computer. It can be complex for someone who is new to programming.

One way to get Jupyter (or something like it) on your computer is to use Google Colab.

3 What is a programming language? What is Python?

A general-purpose computer can be given instructions as to what it should do. Those instructions come in the form of binary numbers (1s and 0s). You can, in theory, write those 1s and 0s yourself – but it's not fun or easy.

Instead, we write programs in a "high-level language," one that is closer to how humans think and write, and then our program is translated into binary format.

Every programming language works in a slightly different way. Some are closer to the binary code, and some are closer to how humans think and write.

- C is a "low-level language," because it's closer to the hardware
- Java and C# are mid-level languages, because they're higher level than C, taking care of such things as managing the memory in your program, but they still require you to think a lot like a computer
- Python is a "very high level language," where it tries to look and feel like a natural language.

Programming languages need to be unambiguous and clear. Python, though, is designed to be as easy as possible for people to learn, while still providing us with the same power as other languages. "Low floors and high ceilings"

Who is using Python, and for what? - Python is the #1 language for data science and machine learning - Web applications - Devops - APIs (consumption and production) - Education

The one place where Python is missing (sort of) is on mobile apps. But that is (slowly) changing.

What's the drawback of Python? It doesn't execute as quickly as Java, C#, or even C. Python is a perfect language for an age in which people are expensive, and computers are cheap.

4 Quick intro to Jupyter

Everything in Jupyter is done inside of "cells." I'm typing, right now, into a cell.

Every cell has a "mode," typically either Python code or Markdown. (Right now, I'm in Markdown mode.) Markdown is used for creating HTML/stylized text easily, with just plain text.

When I'm in Markdown mode, I'm just typing. I can format it by using shift+ENTER. (Some computers, control-ENTER, some alt-ENTER.)

```
[1]: # the below line is Python code; we're running the "print" function, and␣
      ↪telling it to display text
```

```
# anything with a # before it in Python is a comment. It is completely ignored␣
 ↪by the language
# comments are notes to yourself in the future, or to colleagues who will want␣
 ↪to read your code.

print('Hello, world!')

# how can I execute this code cell? I press shift+ENTER
```

```
Hello, world!
```

5 What did we do above?

- `print` is a function – a verb – in the Python world. It displays things on the screen.
- To execute the function, and get it to do something, we put parentheses after its name.
- We can ask it to print text by putting that text inside of the parentheses
- Note that text must be inside of quotes, either `''` or `""`. (You can use either; there is no difference, but the start quotes must match the end quotes.)

[2]:
```
print('Reuven')
```

```
Reuven
```

[3]:
```
# I can even print numbers
print(50)
```

```
50
```

[4]:
```
print(50+3)    # here, I add 50+3, getting 53, and that number is then displayed␣
 ↪on the screen
```

```
53
```

[5]:
```
# numbers don't have quotes around them
# text does

# what if I add together something slightly different:
print('50' + '3')    # can I add together two pieces of text with +?
```

```
503
```

[7]:
```
# What happens, by  the way, if I try to add together a number and text?

# will Python turn the number into text, and give us '503'?
# will Python turn the text into a number, and give us 53?
# or will it give up, and give us an error?  *** this is the answer
print(50 + '3')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[7], line 6
      1 # What happens, by  the way, if I try to add together a number and text?
      2
      3 # will Python turn the number into text, and give us '503'?
      4 # will Python turn the text into a number, and give us 53?
      5 # or will it give up, and give us an error?  *** this is the answer
----> 6 print(50 + '3')

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

[11]:
```
print(50 + 3)
    print(50 + 3)
print(50 + 3)
```

```
  Cell In[11], line 2
    print(50 + 3)
    ^
IndentationError: unexpected indent
```

[12]:
```
# what if I want to keep values around, rather than typing them explicitly at␣
↪the computer?
# we can assign a value to a variable. Then we can refer to that value with the␣
↪variable.

# assignment is done with the = sign. This is *NOT* the same = as you use in␣
↪mathematics!
# In math, = means that the left side is the same as the right side.
# In Python, = means that we take the value from the right side, and assign it␣
↪to the variable on the left side.

x = 10
y = 20

# after executing the above lines, we can refer to x and y, rather than to 10␣
↪and 20

print(x+y)
```

```
30
```

[13]:
```
# by the way, in Jupyter (not in regular Python code!), if the final line of a␣
↪cell is
# a value, then we don't need to use print to display it. Jupyter will just␣
↪show it to us

x
```

[13]:
```
10
```

[14]:
```
x+y
```

[14]:
```
30
```

6 Variables

1. What names can we give our variables?

- Any combination of letters, digits, and _ is acceptable
  - Except: Don't start with a digit (not allowed)
  - Except: Don't start with _ (per convention)
- Capital and lowercase letters are different! By tradition, in Python, we only use lowercase letters in our variable names
- Try to use variable names that mean something; Python couldn't care less what you call your variables, but your colleagues will care a lot!

2. Don't we need to declare them at some point?

No! The first time you assign to a variable, it is created. Next times, the value will be assigned to that variable.

In some languages, you need to declare variables in advance, to tell the language what type of value it will contain. Python variables can refer to any type of value at all! The notion of declaring them doesn't really exist.

[15]:
```
name = 'Reuven'

print('Hello, ' + name + '!')
```

```
Hello, Reuven!
```

[16]:
```
x = 10
y = 20

print(x + y)
```

```
30
```

7 Exercise: Simple assignment and displaying

1. Assign the variable `name` to your name, and print a nice greeting to yourself.

2. Assign two numbers to `x` and `y`, and print the result of adding them together with `+`.

[17]:
```
# if you're in Jupyter, you can use the "magic command" of %whos to get a list␣
↪of all assigned variables

%whos
```

```
Variable   Type    Data/Info
----------------------------
name       str     Reuven
x          int     10
y          int     20
```

[18]:
```
name = 'Reuven'    # don't forget that text needs to have quotes around it --␣
↪variables *never* have quotes around them!

print('Hello, ' + name + '!')
```

```
Hello, Reuven!
```

[20]:
```
print('Hello,' + name + '!')  # no space at the end of the first string
```

```
Hello,Reuven!
```

[22]:
```
greeting = 'Hello, ' + name + '!'    # assign a variable the value of our new␣
↪text string
print(greeting)
```

```
Hello, Reuven!
```

[23]:
```
x = 15
y = 238
print(x + y)
```

```
253
```

[24]:
```
# we can always ask Python what type of value is in a variable (or what type of␣
↪value something is)
# we can use the "type" function

type('Reuven')
```

[24]:
```
str
```

[25]:
```
type(name)   # we assigned the variable 'name' to the text string 'Reuven'
```

[25]:
```
str
```

[26]:
```
print("Hello, " + name + "!")
```

```
Hello, Reuven!
```

```
[27]: type(x)
```

```
[27]: int
```

```
[28]: print('hello')  # this works fine
```

```
hello
```

```
[29]: Print('hello')   # why does this not work?
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[29], line 1
----> 1 Print('hello')   # why does this not work?

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

```
[30]: # what if I want to display numbers and text together?
      # we've seen that this will fail in a big way.

      x = 10
      y = 20
      total = x + y

      # never mind that this code is super ugly!
      # it also won't work!
      print('When you add ' + x + ' and ' + y + ', you get ' + total + '.')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[30], line 10
      6 total = x + y
      8 # never mind that this code is super ugly!
      9 # it also won't work!
---> 10 print('When you add ' + x + ' and ' + y + ', you get ' + total + '.')

TypeError: can only concatenate str (not "int") to str
```

```
[31]: # we can solve this problem by using a special kind of string
      # known as an "f-string"
      # basically: Put an f before the opening quote
      # then inside of the string, you can put {}, and inside of those, you can put␣
       ↪Python values
      # they will automatically be turned into strings, if they aren't already

      print(f'When you add {x} and {y}, you get {total}.')
```

```
When you add 10 and 20, you get 30.
```

[32]:
```python
# we can do even better!

print(f'When you add {x} and {y}, you get {x+y}.')
```

```
When you add 10 and 20, you get 30.
```

8 Next up

- Getting input from the user
- Equality and comparisons
- `if` and making decisions in our programs

[33]:
```python
# how can we get input from the user?
# answer: the input function!

# the input function takes an argument, a text string -- the question that will␣
 ↪be presented to the user
# whatever the user enters is returned by the input function -- and is returned␣
 ↪as a text string
# normally, we'll then take that input and assign it to a variable

# in assignment in Python, *ALWAYS* the right side executes before the left
# here, the input function will run, and wait for the user's input
# whatever the user types will be assigned to the name variable
name = input('Enter your name: ')
```

```
Enter your name:  Reuven
```

[34]:
```python
print(name)
```

```
Reuven
```

[35]:
```python
type(name)
```

[35]:
```
str
```

[36]:
```python
favorite_number = input('Enter your favorite number: ')
```

```
Enter your favorite number:  72
```

[37]:
```python
# remember: input always returns a text string
# it so happens that in Python, multiplying a text string by an integer gives␣
 ↪you a new, longer string

print(favorite_number * 2)   # what will happen?
```

```
7272
```

[38]:
```
# we can think of a function as something that calculates a value and then␣
 ↪"returns it" to us.
# so far, we haven't seen much of that, because print doesn't really return␣
 ↪something -- it displays on the screen.
# most functions are invoked to get their value back

# input's value is whatever the user typed.

# another function is "len", which tells us the length of a string
len('abcd')
```

[38]:
```
4
```

[39]:
```
# because len('abcd') returns 4
# and because we're assigning that value to length_of_the_string

length_of_the_string = len('abcd')
```

[40]:
```
print(length_of_the_string)
```

```
4
```

9 Exercise: Friendly greeting

1. Ask the user to enter their name, and assign that to the variable `name`.
2. Ask the user to enter their country, and assign to the variable `country`.
3. Print a friendly greeting, showing both the name and the country.

[41]:
```
name = input('Enter your name: ')
country = input('Enter your country: ')

print(f'Hello, {name} from {country}!')
```

```
Enter your name:  Reuven
Enter your country:  Israel

Hello, Reuven from Israel!
```

[42]:
```
# we've seen at least one operator, namely +, which lets us add things together
# the result of + is a new integer or a new text string

# there are a bunch of other operators, comparison operators, that return True/
 ↪False
# they are around to tell us whether things are the same or different.
# they are:

# == -- this is the "equality" operator. It returns True if the things on both␣
 ↪sides are the same
```

```
x = 10
y = 10

x == y   # this returns True/False
```

```
[42]: True
```

```
[43]: x = 10
      y = 20

      x == y
```

```
[43]: False
```

```
[44]: x = 10
      y = '10'   # text string 10

      x == y
```

```
[44]: False
```

10 Comparison operators

- `==` – equality, returns `True` if they are equal in value
- `!=` – inequality, returns `True` if they are *not* equal in value
- `<` – less than
- `>` – greater than
- `<=` – less than or equal
- `>=` – greater than or equal

Note that `!=` is one operator, all together. It is *not* the `!` operator that flips the logic of whatever comes to its right.

11 Conditionals

The core of all programming is the ability to make a decision based on data. If something is `True`, then we want to do one thing. If it's `False`, then we want to do another thing.

The way that we make these decisions is with "conditionals."

12 How do we write conditionals?

1. We start with `if`. To the right of `if` is an expression that returns `True` or `False`.
    - No parentheses are needed around the condition
    - At the end of the line, you *must* have a colon (`:`)
2. Following `if`, you have one or more lines that are indented, known as a "block"
    - Must have at least one line

    - Indentation can be any combination of tabs and spaces, *but* you must be consistent; the Python world has generally standardized on 4 spaces
    - If you're typing on the space bar to indent, then you're doing something wrong – your tools for writing Python should know how to indent
    - To finish indentation, just use backspace
3. You can then have an `else`, which is optional
    - Must have a colon at the end of the line
    - This block executes if the `if`'s expression was `False`
    - Another indented block, just like the previous one

If you use `if`/`else`, then one – and only one! – of those blocks is guaranteed to run.

[47]:
```python
name = input('Enter your name: ')

if name == 'Reuven':
    print('Hello, boss!')
    print('It has been far too long!')
else:
    print(f'Hello, {name}. Who are you?')
```

```
Enter your name:  someone else

Hello, someone else. Who are you?
```

13 Exercise: Which word comes first?

1. Ask the user to enter two words, each with its own call to `input` and assigned to a different variable. (Use `first` and `second` as the variables.)
2. Indicate which of the words comes earlier alphabetically.

Hints/tips: - Let's assume that the user will *not* enter the same word twice. - Also assume that the user will enter only lowercase letters, no punctuation/spaces/etc. - You can use `<` and `>` to compare text strings, not just numbers. Whatever comes earlier alphabetically is considered to be "less than" the other.

[49]:
```python
first = input('Enter first word: ')
second = input('Enter second word: ')

if first < second:
    print(f'{first} comes before {second}')
else:
    print(f'{second} comes before {first}')
```

```
Enter first word:  soup
Enter second word:  dessert

dessert comes before soup
```

[50]:
```python
# RM
```

```
first = input('Enter first word: ')
second = input('Enter second word: ')

if first < second:
    print(f'{first} comes before {second}')
else:
    print(f'{second} comes before {first}')

if first > second:
    print(f'{first} comes after {second}')
else:
    print(f'{second} comes after {first}')
```

```
Enter first word:  dessert
Enter second word:  soup

dessert comes before soup
soup comes after dessert
```

[52]:
```
# JG
print (first + " comes first")
```

```
dessert comes first
```

[54]:
```
# how does Python compare text strings?

# it compares the first letter of each string. If the letters are different,␣
 ↪then it declares a winner
# if not, then it compares the second letters. If the letters are different,␣
 ↪then it declares a winnner
#. ...
# it keeps doing this until:
# - the strings are identical
# - one ends earlier than the other
# - it compares all of the characters
```

[55]:
```
# what about an additional comparison?
# that is: What if we don't have just two options, but there are more options␣
 ↪than that?
# we can use an "elif" clause
# elif is basically "elseif" -- it comes between the "if" and any "else"
# elif has its own comparison
# you can have as many "elif" clauses as you want
# the first one that has a True condition executes; the rest are ignored.

x = 50

if x > 100:
    print('Greater than 100!')
elif x > 60:
    print('Greater than 60!')
elif x > 40:
    print('Greater than 40!')
elif x > 0:
    print('Greater than 0!')
else:
    print('Very sad; it is <= 0')
```

```
Greater than 40!
```

[56]:
```
# notice that the if/elif/else clauses are checked *in order*
# this means that the order matters!



x = 50

if x > 0:
    print('Greater than 0!')
elif x > 40:
    print('Greater than 40!')
elif x > 60:
    print('Greater than 60!')
elif x > 100:
    print('Greater than 100!')
else:
    print('Very sad; it is <= 0')
```

```
Greater than 0!
```

[59]:
```
name = input('Enter your name: ')

if name == 'Reuven':
    print('Hi, boss!')
elif name == 'someone else':
    print('That is an odd name')

print('Done')
```

```
Enter your name:  Hi there
```

14 Exercise: Which comes first? (with a tie-breaker)

Repeat the previous exercise, but now tell the user if the two words are the same (rather than which comes first).

[62]:
```
first = input('Enter first word: ')
second = input('Enter second word: ')

if first < second:
    print(f'{first} comes before {second}')
elif first > second:
    print(f'{second} comes before {first}')
else:
    print(f'{first} and {second} are the same!')
```

```
Enter first word:  telephone
Enter second word:  Telephone

Telephone comes before telephone
```

[ ]:
```

```

15 Next up

- or/and/not – making our comparisons more sophisticated
- Working with numbers (integers and floats)
- Working with text strings
- String methods

[63]:
```
# sometimes, we don't want to check just one thing
# sometimes, we want to know if A *and* B are both true
# or, in some cases, we want to know if A *or* B is true

# we can do that in Python with
# - and
# - or
# - not (which flips the logic of whatever is to its right, from True -> False␣
 ↪or False -> True
```

[64]:
```
x = 10
y = 20

x == 10
```

[64]: True

[65]:
```
y == 20
```

[65]: True

[66]:
```
#      True     and      True   ---> True
if     x == 10   and    y == 20:
    print('Yes, both are what you want!')
```

```
Yes, both are what you want!
```

[67]:
```python
#       False      and     True    -->  False
if     x == 100    and    y == 20:
    print('Yes, both are what you want!')
```

[68]:
```python
# or means: if one of them is True, then it's True

#       False      or     True    --> True
if     x == 100    or    y == 20:
    print('Yes, at least one is what you want!')
```

```
Yes, at least one is what you want!
```

[70]:
```python
# in general, one line in Python = one statement/command/expression
# what if you want to split something across lines?

# if you have open parentheses, then Python is much more flexible about this␣
 ↪sort of thing
# still-open parentheses are treated as a continuation of the previous line

if (x == 100 or
    y == 20):
    print('Yes, at least one is what you want!')
```

```
Yes, at least one is what you want!
```

[71]:
```python
# not is used more rarely, but we can use it

x = 10

if not x == 20:
    print('It is not 20')
```

```
It is not 20
```

16 Exercise: Name and company

1. Ask the user to enter their name, and assign to the variable `name`.
2. Ask the user to enter their company, and assign to the variable `company`.
3. Print one of four things:
    - If the name and company are the same as you, then greet yourself
    - If the name is the same, but the company is different, greet someone else with a great name
    - If the company is the same, but the name is different, greet your colleague
    - If both name and company are different, give a snarky greeting.

[ ]:
```python

```

[72]:
```
# don't say "if not ... == " rather say !=
```

[73]:
```
name = input('Enter your name: ')
company = input('Enter your company: ')

if name == 'Reuven' and company == 'Lerner':
    print('Hey, you are me!')
elif name == 'Reuven':
    print('Great name, person from another company!')
elif company == 'Lerner':
    print('Hello, my colleague!')
else:
    print('Hello, weirdo with a terrible name and a terrible employer!')
```

```
Enter your name:  asdfsafafdsa
Enter your company:  asdfafafsafsafas

Hello, weirdo with a terrible name and a terrible employer!
```

[74]:
```
# some programming languages let you use <> for "not equal" -- not in Python,␣
 ↪where it's !=
```

[75]:
```
# Let's talk about numbers in Python!

# int -- integers, whole numbers
# float -- numbers with a decimal point

x = 10
type(x)
```

[75]:
```
int
```

[76]:
```
x = 10
y = 3
```

[77]:
```
# what operations can I do on these?

x + y    # addition
```

[77]:
```
13
```

[78]:
```
x - y # subtraction
```

[78]:
```
7
```

[79]:
```
x * y    # multiplication
```

[79]:
```
30
```

```
[81]: x / y    # division  -- we get a floating-point value back
```

```
[81]: 3.3333333333333335
```

```
[82]: x // y    # floor division -- we get the answer as an integer, chopping off any␣
      ↪fractional part
```

```
[82]: 3
```

```
[83]: x ** y    # exponentiation
```

```
[83]: 1000
```

```
[85]: x % y  # "modulo" -- the integer remainder after dividing x/y
```

```
[85]: 1
```

```
[86]: # what if I want to add 1 to an integer stored in a variable?

      x = 10
      x = x + 1    # notice, this is only OK because = is not the math =, but rather␣
      ↪means "assign"
      x
```

```
[86]: 11
```

```
[87]: # shorthand for this:

      x = 10
      x += 1    # this means: x = x + 1
      x
```

```
[87]: 11
```

```
[88]: # remember, strings and integers cannot be added together

      x = 10
      y = '20'

      x + y
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[88], line 6
      3 x = 10
      4 y = '20'
----> 6 x + y
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

[89]:
```python
# I can get a new integer based on y by calling int() on y

int(y)  # this returns a new integer -- it doesn't change y!
```

[89]:
```
20
```

[90]:
```python
# one solution is to say:

x + int(y)
```

[90]:
```
30
```

[91]:
```python
# I could also say
y = int(y)
x + y
```

[91]:
```
30
```

17 Guessing game

1. Assign an integer to the variable `secret`. This is the number that the user needs to guess.
2. Ask the user to guess the secret integer.
3. Give the user one of these three outputs:
    - Too high
    - Too low
    - You got it

The user gets *one* chance to guess the number!

[92]:
```python
secret = 42

guess = input('Enter your guess: ')

if guess < secret:
    print('Too low!')
elif guess > secret:
    print('Too high!')
else:
    print('You got it!')
```

```
Enter your guess:  42
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[92], line 5
```

```
      1 secret = 42
      3 guess = input('Enter your guess: ')
----> 5 if guess < secret:
      6     print('Too low!')
      7 elif guess > secret:

TypeError: '<' not supported between instances of 'str' and 'int'
```

[93]:
```
42 == '42'
```

[93]:
```
False
```

[97]:
```
secret = 42

guess = input('Enter your guess: ')
guess = int(guess)    # turn guess into an integer

if guess < secret:
    print('Too low!')
elif guess > secret:
    print('Too high!')
else:
    print('You got it!')
```

```
Enter your guess:  hello out there
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[97], line 4
      1 secret = 42
      3 guess = input('Enter your guess: ')
----> 4 guess = int(guess)    # turn guess into an integer
      6 if guess < secret:
      7     print('Too low!')

ValueError: invalid literal for int() with base 10: 'hello out there'
```

[98]:
```
# floats

# if a number contains a decimal point, it's known as a "float"
# you can convert a float to an int , or an int, to a float

x = 10
float(x)
```

[98]:
```
10.0
```

```
[99]: x = 12.34
      type(x)

[99]: float

[100]: int(x)   # get an integer based on the float

[100]: 12

[101]: # if you use floats and ints together in the same expression, then that's fine␣
        ↪-- the int is
       # "promoted" to a float

       10 + 20.35

[101]: 30.35
```

18 Next up

- Strings
- Defining them
- Retrieving from them
- Searching in them
- Strings are *immutable*
- A little about methods

```
[ ]: # GM
     TypeError                                 Traceback (most recent call last)
     <ipython-input-57-c1f56655dbca> in <cell line: 2>()
           1 secret = 5
     ----> 2 guess = input('What is your number? ')
           3 guess = int(guess)
           4
           5 if guess < secret:

     TypeError: 'int' object is not callable

[102]: # when we invoke a function, that's also known as "calling" a function
       # functions are known as "callable," meaning that we are able to call them

       # it's saying here that you tried to call an integer, and ints aren't callable

       x = 5
       x()

       ---------------------------------------------------------------------------
       TypeError                                 Traceback (most recent call last)
```

```
Cell In[102], line 7
      1 # when we invoke a function, that's also known as "calling" a function
      2 # functions are known as "callable," meaning that we are able to call␣
 ↪them
      3
      4 # it's saying here that you tried to call an integer, and ints aren't␣
 ↪callable
      6 x = 5
----> 7 x()

TypeError: 'int' object is not callable
```

[103]:
```
# it's very easy to accidentally assign a new value to many of Python's builtin␣
 ↪functions
# you must have accidentally assigned an integer to input, as in

# input = int('x')

# how do you fix this?
# (1) restart Python -- go to the "Kernel" menu at the top of the screen (in␣
 ↪Jupyter) and click on "restart"
# (2) say "del input" in Python, which will delete the version you created, and␣
 ↪restore access to the original
```

19 Strings

In Python, no matter how long or short your text is, it's going to be in a "string." A string can contain any number of characters from any character set (language) in the world.

You define a string with quotes (either single or double).

[104]:
```
s = 'abcdefghijklmnopqrstuvwxyz'

len(s)    # how many characters are in s?
```

[104]:
```
26
```

[105]:
```
# how can I get the first character from our string?
# answer: Use [], with the index I want inside of them
# note: the indexes start with 0, not 1

s[0]    # this means: give me the first character in s
```

[105]:
```
'a'
```

[106]:
```
s[1]   # this means: give me the second character in s
```

```
[106]: 'b'
```

```
[107]: # how can I get the final character in s?
       s[25]   # remember, if there are 26 letters, and the first is at index 0, the␣
        ↪final one is at index 25
```

```
[107]: 'z'
```

```
[108]: # another technique: Calculate the final one
       i = 25
       s[i]   # this is totally fine
```

```
[108]: 'z'
```

```
[109]: i = len(s) - 1   # get the length, and subtract 1
       s[i]   # this is the final character, too
```

```
[109]: 'z'
```

```
[110]: # inside of square brackets, I can do something similar:
       s[ len(s)-1 ]
```

```
[110]: 'z'
```

```
[111]: # we can also use negative indexes to get from the right.
       s[-1]
```

```
[111]: 'z'
```

```
[112]: s[-2]
```

```
[112]: 'y'
```

```
[114]: s[-3]
```

```
[114]: 'x'
```

```
[115]: s[100]   # what about asking for something that's too big?
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[115], line 1
----> 1 s[100]   # what about asking for something that's too big?

IndexError: string index out of range
```

```
[116]: # what if I want to search in my string?
       # I can use the "in" operator

       'j' in s
```

```
[116]: True
```

```
[117]: 'a' in s
```

```
[117]: True
```

```
[118]: '!' in s
```

```
[118]: False
```

```
[119]: 'bcd' in s   # this checks if the entire string, 'bcd', as written, is in s
```

```
[119]: True
```

```
[120]: 'bd' in s
```

```
[120]: False
```

```
[121]: # I can use a slice -- to get a part of the string
       # this uses the syntax [start:end]
       # we get back a new string based on s, starting at "start" and ending *before*␣
        ↪"end"

       s[10:20]   # this returns s[10] up to , and not including, s[20]
```

```
[121]: 'klmnopqrst'
```

```
[122]: s[10:25]  # from s[10] up to and not including s[25]
```

```
[122]: 'klmnopqrstuvwxy'
```

```
[123]: s[:10]   # this means: up to, and not including, s[10]
```

```
[123]: 'abcdefghij'
```

```
[124]: s[20:]   # this means, from s[20] through the end
```

```
[124]: 'uvwxyz'
```

```
[125]: # strings are immutable - -they cannot be changed!

       x = 'abcde'
       x[0]
```

```
[125]: 'a'
```

```
[126]: x[0] = '!'    # can I modify the string?
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[126], line 1
----> 1 x[0] = '!'    # can I modify the string?

TypeError: 'str' object does not support item assignment
```

```
[127]: s
```

```
[127]: 'abcdefghijklmnopqrstuvwxyz'
```

```
[128]: x
```

```
[128]: 'abcde'
```

```
[129]: # on the right side of the assignment, I create a new string
       # I then assign that new string to x
       # x then refers to the new string, not to 'abcde'
       # but we have *not* changed a string, we've merely said that x refers to a new,␣
        ↪different one.
       x = x + 'fghij'
```

```
[130]: i = 10

       # can I now change the value of 10 to be 8?
```

20 Exercise: Pig Latin translator

If you aren't yet familiar with Pig Latin, then: - It's a children's "secret" language - You can translate a word from English into Pig Latin by checking the first letter: - If it's a vowel (a, e, i, o, u) then we add `way` to the word - Otherwise, we move the first letter to the end, and add `ay`

Examples: - `computer` -> `omputercay` - `octopus` -> `octopusway` - `table` -> `abletay` - `elephant` -> `elephantway` - `papaya` -> `apayapay`

- You are to ask the user to enter a word (all lowercase, no punctuation, no spaces).
- Print the translation into Pig Latin.

```
[132]: word = input('Enter a word: ')

       if word[0] == 'a' or word[0] == 'e' or word[0] == 'i' or word[0] == 'o' or␣
        ↪word[0] == 'u':
           print(word + 'way')
```

```
Enter a word:  table
```

[137]:
```python
# isn't there a better way?
# yes, there is -- and it's not obvious to most people, at least at first

word = input('Enter a word: ')

if word[0] in 'aeiou':
    print(word + 'way')
else:
    print(word[1:] + word[0] + 'ay')
```

```
Enter a word:  hooray
```

```
oorayhay
```

21 Methods

So far, all of the verbs we've encountered in Python are functions:

- `print`
- `input`
- `len`

When we want to invoke a function, we do that with `()`:

- `print('abcd')`
- `input('Enter your name: ')`
- `len('abcd')`

But most of the verbs in Python are actually not functions. They are *methods*, a term from object-oriented programming. If you're working with strings or most other data structures in Python, then most of the things you invoke will be methods.

They look a little different. Instead of

`FUNCTION(DATA)`

you instead say

`DATA.METHOD()`

Strings have a lot of methods:

- `str.lower` – this returns a new string, containing only lowercase letters
- `str.upper` – this returns a new string, containing only uppercase letters

[138]:
```python
s = 'aBcD eFgH'

s.lower()
```

[138]: `'abcd efgh'`

```
[139]: s.upper()
```

```
[139]: 'ABCD EFGH'
```

```
[140]: s
```

```
[140]: 'aBcD eFgH'
```

```
[141]: # here's one of the most important methods:

       name = input('Enter your name: ')
       print(f'Hello, {name}!')
```

```
Enter your name:              Reuven

Hello,            Reuven           !
```

```
[142]: name
```

```
[142]: '           Reuven           '
```

```
[143]: # the "strip" method removes whitespace from the start and end of a string (not␣
         ↪inside)
       # it returns a new string without the outer whitespace

       name.strip()
```

```
[143]: 'Reuven'
```

```
[144]: name = input('Enter your name: ')
       name = name.strip()

       print(f'Hello, {name}!')
```

```
Enter your name:        Reuven

Hello, Reuven!
```

```
[145]: # we can do even better!
       # we can call str.strip on *any* string
       # this includes the string we get back from input

       name = input('Enter your name: ').strip()

       print(f'Hello, {name}!')
```

```
Enter your name:      Reuven

Hello, Reuven!
```

[147]:
```
# we've been asking the user for integer inputs

secret = 42

guess = input('Enter your guess: ')
guess = int(guess)    # turn guess into an integer

if guess < secret:
    print('Too low!')
elif guess > secret:
    print('Too high!')
else:
    print('You got it!')
```

```
Enter your guess:  what?
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[147], line 6
      3 secret = 42
      5 guess = input('Enter your guess: ')
----> 6 guess = int(guess)    # turn guess into an integer
      8 if guess < secret:
      9     print('Too low!')

ValueError: invalid literal for int() with base 10: 'what?'
```

[150]:
```
# we can use the method str.isdigit
# this returns True if the string contains only digits 0-9
# if there any non-digit there, it returns False
# (it also returns False for an empty string, with 0 characters)

# we've been asking the user for integer inputs

secret = 42

guess = input('Enter your guess: ')

if guess.isdigit():    # isdigit is a *string* method
    guess = int(guess)    # turn guess into an integer

    if guess < secret:
        print('Too low!')
    elif guess > secret:
        print('Too high!')
    else:
        print('You got it!')
else:
    print(f'{guess} is not numeric! You lose!')
```

```
Enter your guess:  I don't know

I don't know is not numeric! You lose!
```

[152]:
```
# how can we know what methods exist for strings?
# (1) In Jupyter and many editors, you can type . after a string and then tab,␣
 ↪and get a menu

s = 'abcd'

# (2) https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str
```

[153]:
```
s = 'aBcD eFgH'

s.lower()
```

[153]:
```
'abcd efgh'
```

[154]:
```
s.upper()
```

[154]:
```
'ABCD EFGH'
```

[155]:
```
s.capitalize()
```

[155]:
```
'Abcd efgh'
```

[156]:
```
s.title()
```

[156]:
```
'Abcd Efgh'
```

[157]:
```
# my favorite dumb method is.. swapcase!

s.swapcase()
```

[157]:
```
'AbCd EfGh'
```

[ ]:
```

```