🔤

Day 1 Cheat Sheet

The following cheat sheet was written by Stacy Li for the CCB Python Bootcamp.
Personal usage is permitted, but do not copy or redistribute without express permission from SL (stacy_li (αt) berkeley (dot) edu).

💡 We recommend that you turn on dark mode with `Command/Ctrl + Shift + L` for the best viewing experience.

Table of Contents

PDF version

If you would prefer to download or print this cheat sheet, click the link below.

🖨️ [CLICK ME] Day 1 Cheat Sheet

Data types & variables

Representations of data and desired analyses in Python are encoded as **objects**. Each object has its own **type**, which may influence the operations you can perform with/on that object.

Information can be represented in different ways in Python, classified according to **data types**.

Type	Description	Examples
string (`str`)	Strings of characters. Prose and other text are represented as strings.	`"TATAAA" 'TATAAA' ''' TATAAA ''' "6" "6.0" "None"`
integer (`int`)	Integer values of numbers.	`6`
float (`float`)	Numbers with decimals ("floating point").	`6.0`
None (`NoneType`)	A special type that is akin to a NA (not available) value. It represents the absence of information.	`None`

Variables represent data. Each variable will thus have a type in accordance with the data it represents.

Task	Operation
Assigning a variable	Variable name on the left, followed by an `=` sign, followed by the value (data). Example: `tata_box = TATAAA`
Updating/reassigning a variable	Same as assigning a variable. Example: `x = 5` followed by `x = 6` updates the value of `x` to 6.
"Calling" a variable.	Enter the variable name. Example: `tata_box` will return `TATAAA`, the value "within" `tata_box`.
Check variable contents	Either use `print()` or call the variable directly.

Built-in functions

Functions perform a desired task in Python, sometimes requiring an **input.** We've listed some useful **built-in functions** below.

Task	Operation
"Calling" a function	Enter the function name with parentheses enclosing the input, if there is one. Example: `sum(5, 3)`
Learning what the function does	Use the `help()` function.
Printing values (strings, numerics, etc) to output	Use the `print()` function.
Finding the length of an object	Use the `len()` function.

Basic operations

Numerics

You can perform simple mathematical operations in Python without importing new modules/packages. (More on that on Day 2!)

> 💡 Remember that order of operations applies. Use parentheses to encapsulate your operations.

Task	Operation
Addition	`5 + 3`
Subtraction	`5 - 3`
Multiplication	`5 * 3`
Raising to an exponent	`5**3`: equivalent to 5 to the power of 3.

Task	Operation
Division (always yields float)	`> 1/3 0.3333333` `> 15/3 5.0`
Division (yields lowest whole integer value)	Use double slashes (`//`), also known as the **floor** operator. `>15//4 3`
Division (get remainder only)	Use the percent sign (`%`), also known as the **modulo** operator. `>25%20 5`

Strings & methods for strings

Strings can also be manipulated in elegant ways without having to import external modules/packages. (More on that on Day 2!)

Substrings refer to shorter strings within a “main” string. For example, `'AUG'` is a substring of `'AUGCUGAUUGAC'`.

Task	Operation
Get the length of a string	Use the `len()` function.
Concatenation (addition)	Combines strings end-to-end in the order provided. `> “5’-GATT” + “ACA-3’” ”5’-GATTACA-3’”`
Repetition	Repeats content of a string. `> “Hello! “ * 3 ”Hello! Hello! Hello!"`
Join strings with a specific substring	Use the `.join()` method. (If you provide an empty string (`’’`), the strings will be joined without anything in-between.) Example: `''.join(['abcd', 'efgh'])` returns `'abcdefgh'`. `' '.join(['Hello', 'world']` returns `'Hello, world'`.
Split a string at a specific substring	Use the `.split()` method. Example:`'abcde'.split('c'` returns `[’ab’, ‘de’]`.
Replace specific substrings within a string	Use the `.replace()` method. Example: `'TAAGTACAG'.replace('AG', 'CT')` returns `'TACTTACCT'`.
Check if a substring is contained within a larger string.	Use the `in` operator. Example: `'a' in 'apple'` returns `True`.
Find the first instance (index value) of the element in the string.	Use the `.index()` method. Examples: `'TAAGTACAG'.index('AA')` returns `1`.

For a complete list of methods for strings, click here.

You can use special characters in your strings to format the text displayed with the `print()` function.

Special character	
`\n`	Denotes a line break/newline.
`\t`	Denotes a tab.

Converting data types (coercion)

Coercion refers to explicit re-typing of data types, also known as **typecasting**.

Task	Operation
Integer to string	`str(6)`
Float to string	`str(5.0)`
String to integer	`int("6")`
String to float	`float("6.0")`
Integer to float	`float(6)`

Basic data structures

Lists & methods for lists

Lists are exactly what they sound like: they are containers that house elements in a given sequence/order.

Lists can contain multiple types at the same time. They can also be **nested**, meaning that lists can contain more lists.

```
number_list = [1, 2, 3, 4, 5]
string_list ['a', 'p', 'p', 'l', 'e']
mixed_list = ['orange', 22, 'f', 67.2]
nested_list = [['apple', 'banana'], ['onion', 'potato']]
```

Task	
Get the length of a list	Use the `len()` function.
Get the maximum value in a list	Use the `max()` function. This returns max numerical value or longest string. Example: `max([1,2,3])` returns `3`. `max([’aaa’, ‘b’, ‘aaaa’])` returns `‘aaaa’`.
Get the minimum value in a list	Use the `min()` function. This returns mininum numerical value or shortest string. Example: `min([1,2,3])` returns `1`.
Check if a certain element or value is in a list.	Use the `in` operator. Example: `'a' in ['a', 'b', 'c']` returns `True`.
Reverse the list (in place)	Use the `.reverse()` method.
Sort the list (in place)	Use the `.sort()` method.
Add an element to the last position of the list (in place)	Use the `.append()` method. Example: `list_example = [1, 2, 3]` followed by `list_example.append(4)` results in `list_example` equalling `[1, 2, 3, 4]`.
Add multiple elements to the last position of the list (in place)	Use the `.extend()` method. Example: `list_example = [1, 2, 3]` followed by `list_example.extend([4, 5, 6])` results in `list_example` equalling `[1, 2, 3, 4, 5, 6]`.
Remove an element from the list (in place)	Use the `.remove()` method. Example: `list_example = [1, 2, 3]` followed by `list_example.remove(3)` results in `list_example` equalling `[1, 2]`.
Count the instances of an element in the list	Use the `.count()` method.
Find the first instance (index value) of the element in the list.	Use the `.index()` method.
Coerce a string to a list	Use the `list()` function.

For a complete list of methods for strings, click here.

Iterable objects

Objects are **iterable** if you can sequentially access (indexing, slicing) and perform an operation (ex. a built-in function) on each element of the object.

Here are some examples of iterable objects:

- **Strings** (each character of a string can be iterated)
- **Lists** (each element of a list can be iterated)
- `range()` objects (useful for ranges of numbers)
- **Tuples**

- **Sets**
- **Dictionaries** (both keys, values, and key-value pairs)

Indexing and slicing

Python employs **zero-based indexing**, meaning that the leading element of any iterable is indexed at `0`.

```
someString:  a b c d e f g h i j
Index:       0 1 2 3 4 5 6 7 8 9
```

```
# showing you these iterables again for reference
number_list = [1, 2, 3, 4, 5]
string_list ['a', 'p', 'p', 'l', 'e']
```

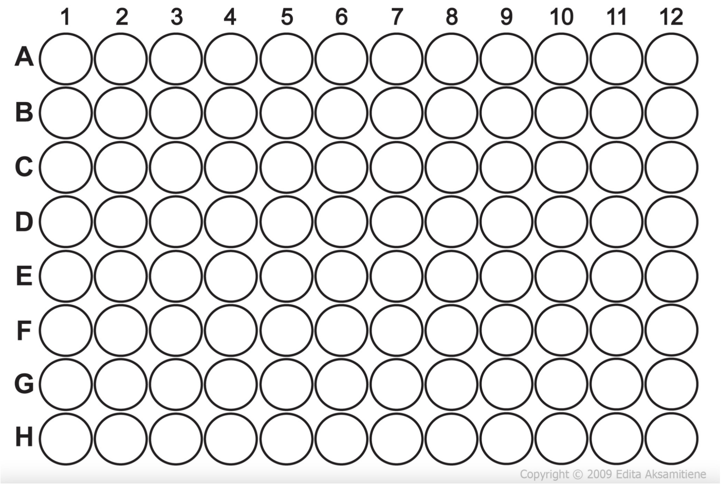
Task	
Accessing a specific element	Use the index of the element. Example: `number_list[1]` returns 2. `'racecar'[3]` returns `'e'`.
Get the first element	Use `[0]`.
Get the last element	Use `[-1]`.
Slicing elements	Use the `:` operator to indicate a “slice”. **Remember that slices are non-inclusive of the last index.** Example: `number_list[1:3]` returns `[2, 3]`. `'racecar'[1:3]` returns `'ac'`.
“Slice” from a specified position through the end of the iterable	If `number_list` is `[1, 2, 3, 4, 5]`: `> number_list[2:] [3, 4, 5]` `> ‘racecar’[1:]` `’acecar’`
“Slice” from the beginning of the iterable until a specified position	If `string_list` is `['a', 'p', 'p', 'l', 'e']`: `> string_list[:-1] ['a', 'p', 'p', 'l']` `>` `'racecar'[:4]` `’race’`

`for` loop

Executes the same code for every element of an iterable. Constructed in the form of:

```
for <ITEM> in <ITERABLE>:
  <EXECUTE CODE HERE>
```

If this is not intuitive, try copy/pasting this example into a code cell and running it.



Copyright © 2009 Edita Aksamitiene

A plain 96 well plate: 12 columns, 8 rows.

```
# Example: Consider a 96 well plate. (image below)
# A 96 well plate has 8 rows (A-H) and 12 columns (1-12).
# If we were "looping" across every instance (well) in a 96 well plate...

rows = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
columns = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12']

for row in rows:
  for col in columns:
    print(row+col)
```

The built-in `range()` function quickly creates an iterable of numbers, saving lots of time compared to manually typing out a list of numbers. Try running the example below:

```
for i in range(0, 5):
  print(i)
```

Boolean logic

With Boolean logic, there are two outcomes of a **logic check**: either it's `True` or it's `False`. These correspond to the two Boolean type objects available in Python.

Value	
`True`	Indicates a "pass" of some Boolean logic check. Examples: `5 == 5` yields `True`. `5 > 3` yields `True`.
`False`	Indicates a "fail" of some sort of Boolean logic check. Example: `5 == 3` yields `False`. `-3 > 5` yields `False`.

Logic and conditional code execution

Logical operators allow you to perform logic checks on your code.

Operator	Meaning
`>`	Greater than
`<`	Lesser than
`>=`	Greater than or equal to

Operator	Meaning
`<=`	Less than or equal to
`==`	“Is equal to?” (checking equality)
`!=`	“Is *not* equal to?” (checking inequality)
`in`	Checks for membership: is the left object present in the right object?

Conditional statements

Conditional statements should be ordered in the form of `if`, `elif` (if using), and `else` (if using).

Statement	
`if`	Code will be executed *if* the logic check passes (returns `True`).
`else`	Code will be executed if the preceding `if` statement’s logic check fails (returns `False`).
`elif`	A secondary statement that must follow the initial `if` statement. If the initial `if` statement’s logic check fails, then the `elif` statement’s logic check will be executed in the same manner as an `if` statement.

```
# the if statement can exist by itself...
if <LOGIC CHECK PASSES>:
  <EXECUTE CODE HERE>

# elif statements are not mandatory, but can be useful if you want something
# else to happen if your initial logic check does not pass
elif <LOGIC CHECK PASSES>:
  <EXECUTE CODE HERE>

# you can write multiple elif statements
elif <LOGIC CHECK PASSES>:
  <EXECUTE CODE HERE>

# else statements are not mandatory, but can be useful to specify some specific
# code that should run if none of the above logic checks pass
else:
  <EXECUTE CODE HERE>
```

Remember: You can construct a conditional statement with just a single `if` statement! `elif` and `else` are only used to add additional complexity when necessary.

```
# an example of a simple conditional statement in a function
# run this to see how it works

def name_length(name):
  if len(name) > 3:
    print("If you see this line, the logic check passed.")

name_length("Phil")
```

Remember: `elif` logic checks are “checked” in the order that they appear. Make sure to use `print()` to check that your “cascade” of conditional statements works correctly~

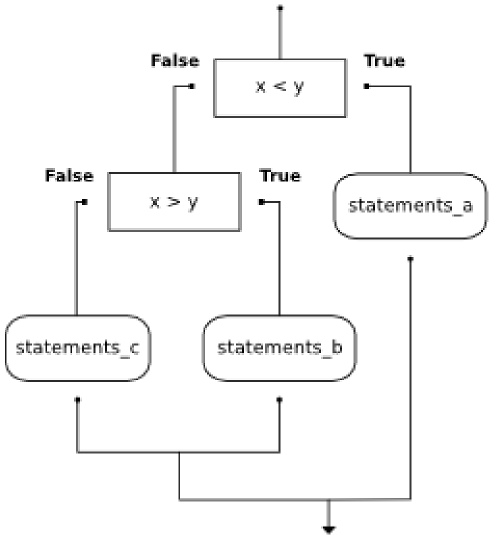
Nested conditional statements

You can *nest* conditional statements in order to add complexity to your logic checks. Below is an example from Runestone:

```
# run this and see what happens
```

```
x = 10
y = 10

if x < y:
    print("x is less than y")
else:
    if x > y:
        print("x is greater than y")
    else:
        print("x and y must be equal")
```



Multiple logic checks

You can use the `and` and `or` operators to add even more complexity to your logic checks.

Operator	
`and`	Indicates that multiple logic checks must be passed to yield `True`. Example: `1 == 1 and 2 == 2` yields `True`. `1 == 1 and 2 == 1` yields `False`.
`or`	Indicates that *at least one* logic check must pass to yield `True`. Example: `1 == 1 or 2 == 2` yields `True`. `1 == 1 and 2 == 1` yields `True`.

```
fruits = ['apple', 'orange', 'banana', 'grapes', 'kiwi']
colors = ['red', 'yellow', 'orange']
sale = ['apple', 'banana']
favorite = ['apple', 'grapes']

for fruit in fruits:
  if (fruit in fruits) and (fruit in colors):
    print("Both a fruit and a color.")
  if (fruit in sale) or (fruit in favorites):
    print("I'll take this today.")
```

Writing your own functions

Use the `def` statement to define your own function.

```
def raise_to_power(a, b):
  # This function raises a to the power of b.
  return a**b
```

Common troubleshooting:

Problem	Solution
`IndentationError`	Make sure that the code you want your function to run is indented! The whole code block must be indented, otherwise it won't work.
Function isn't returning a value	Did you use the `return` statement?
Function is returning the wrong value	Carefully check your code. Also, make sure you're not confusing `print()` with `return`.

`print()` versus `return`

`print()` is a function used for displaying some sort of text or result for human consumption (reading!)

On the other hand, `return` refers to the process of literally *returning* some sort of value to Python for later use. We can deliberately specify this in our functions using the `return` statement, but Python also does this by default for some actions, like calling variables.

For example, when we give Python an object (like a variable containing a string), it returns the value of the variable by default.

```
>>> example_object = "What's up, Python?"
>>> example_object
"What's up, Python?"
```

If we *print* the object, it prints (displays) the value of the object we give it.

```
>>> print(example_object)
What's up, Python?
```

Notice the lack of quotes, because `print()` yields *displayed text*, which is different from a string. The output of `print()` cannot be used for anything else – it's just for display!

Run the below code cell and make sure you understand why the `return` value differs from what is displayed by the `print()` function.

```
def sum_then_square(a, b):
  # This function will coerce a and b to floats, sum the numbers a and b, then square the sum.
  a = float(a)
  b = float(b)
  squared_sum = (a + b)**2
  print(a, b)
  return squared_sum
```

More data structures: tuples, sets, dictionaries

Tuples are *immutable* containers*:* unlike lists, they cannot be altered once created. Tuples are created using `tuple()` or parentheses (`( )`).

```
example_tuple_1 = tuple(['first', 'second'])
example_tuple_2 = ('first', 'second')
```

Tuples gain efficiency at the cost of flexibility. You cannot do the following with tuples:

- Sort, reverse, or otherwise change the order of elements
- Delete elements

- Add elements

You can still:

- Combine existing tuples into a new tuple using the `+` operator.
- Turn a tuple into a list, modify the list, then turn it back into a tuple.
- Use the `.count()` and `.index()` methods in the same manner you would use lists.

For a complete list of methods for tuples, click here.

Sets are literal *sets* of unique values, as classically used in probability. You can create a set using `set()` or curly braces (`{ }`).

```
set_1 = set([1, 2, 3, 3, 4]) # yield same as set_2
set_2 = {1, 2, 3, 4}
set_3 = {1, 2, 3, 4, 5}
set_4 = {5. 6. 7}
```

Like tuples, sets are quite efficient, but at the cost of flexibility.

- Sets are not ordered.
- Sets cannot be indexed.

You can still:

- Add elements to a set
- Remove elements to a set

Method	Description
`.difference()`	Returns a set containing objects that are not found in both sets.
`.intersection()`	Returns a set containing objects found in both sets.
`.union()`	Returns a set with all objects in both sets.
`.issubset()`	Performs a logic check to see if the target set is a *subset* of the input set.
`.issuperset()`	Performs a logic check to see if the target set is a *superset* of the input set.

For a complete list of methods for sets, click here.

Dictionaries are data structures that implement a 1:1 relationship between a **key** and a **value**.

In each dictionary, the *keys* must be unique strings, but the *values* don’t necessarily have to be unique. The values can also contain other data structures, even other dictionaries (forming a **nested dictionary**).

```
dict_1 = {'Michael': 28,
          'Joseph': 25,
            'Anya': 22,
            'Juli': 28,
          'Jennie': 25}

dict_2 = {'living_room': ['TV', 'couch'],
              'bedroom': ('bed', 'dresser'),
              'kitchen': {'fork', 'spoon' 'dish'}}

nested_dict = {'work': {'Michael': 'admin', 'Anya': 'admin', 'Jennie': 'user'},
              'school': {'Joseph': 'teacher', 'Juli': 'teacher'}}
```

Method	Description

Method	Description
`.items()`	Returns the key-value pairs as an iterable of tuples.
`.keys()`	Returns the keys as an iterable.
`.values()`	Returns the values as an iterable.

For a complete list of methods for dictionaries, click here.

Optional

Formatting strings

The `format` method allows you to “form-fill” a string. Use curly brackets with a placeholder name (`{name}`) to indicate where the string should be “filled”.

Run the example from lecture to see how this works:

```
ref_format = '''Type  Name\tRefSeq\tINSDC\tSize (Mb)\tGC%\tProtein
{datatype}\t{name}\t{refseq}\t{INSDC}\t{size}\t{GC}\t{protein}'''

print(ref_format.format(refseq = 'NC_000913.3', INSDC = 'U00096.3',
                       size = 4.64, GC = 50.8, protein = 4242, datatype = 'Chr', name = '-'))
```

You can use special characters to insert line breaks or tabs:

`while` loop

Executes the same code *while* a specified logic check passes (returns `True`). Constructed in the form of:

```
while <LOGIC CHECK RETURNS True>:
  <EXECUTE CODE HERE>
```

If this is not intuitive, try copy/pasting this example into a code cell and running it.

```
# Example: Consider pipetting from a bottle of some reagent.
# You can continue pipetting if there is at least (greater than or equal to) the
# volume of your aliquot remaining in the bottle.

bottle_volume = 1100 # 1000 uL
aliquot = 200 # 200 uL

print(bottle_volume)
while bottle_volume >= aliquot:
  bottle_volume -= 200
  print(bottle_volume)

print("Bottle volume: ", bottle_volume)
print("No more aliquots are possible.")
```

Scope (local/global) and Python Code Visualizer

Variables defined *inside* of a function cannot be accessed *outside* of the function. Below is a command-by-command contents check of a variable called `my_variable`:

```
>>> my_variable = 19
>>> print("my_variable (global) is: ", my_variable)
my_variable (global) is:  19

>>> def my_function(x):
```

```
...     my_variable = x + 5
...     print("my_variable (inside the function) is: ", my_variable)

>>> my_function(10)
my_variable (inside the function) is:  15

>>> print("my_variable (global) is still: ", my_variable)
my_variable (global) is still:  19
```

Not clicking? Use the Python Code Visualizer with the code cell below to see it step-by-step visually.

```
my_variable = 19 # assigned in global
print("my_variable (global) is: ", my_variable)

def my_function(x):
  my_variable = x + 5
  print("my_variable (inside the function) is: ", my_variable)

my_function(10)
print("my_variable (global) is still: ", my_variable)
```

List comprehensions

You can simultaneously iterate through elements in a list *and* create a new list by using a **list comprehension**.

```
# try this out and print the result of fruit_count
fruits = ['apple', 'orange', 'banana', 'grapes', 'kiwi']
fruit_count = [len(x) for x in fruits]
```