PythonCheatSheet

Hello, World!

⇒ Python is object oriented and dynamically type checked.

⇒ With dunder methods, every *syntactic construct* extends to user-defined datatypes, classes! —Including loops, comprehensions, and even function call notation!

⇒ Children block fragments are realised by consistent indentation, usually 4 spaces. No annoying semicolons or braces.

⇒ τ`(x)` to try to coerce `x` into a τ type element, crashes if conversion is not possible. Usual types: `bool`, `str`, `list`, `tuple`, `int`, `float`, `dict`, `set`. Use `type(x)` to get the type of an object `x`.

⇒ Identifier names are case sensitive; some unicode such as "α" is okay but not "⇒".

⇒ If `obj` is an instance of type τ, then we may invoke an instance method `f` in two ways: `obj.f()` or τ`.f(obj)`. The latter hints at why "`self`" is the usual name of the first argument to instance methods. The former τ`.f` is the name proper.

⇒ Function and class definitions are the only way to introduce new, local, scope.

⇒ `del x` deletes the object `x`, thereby removing the name `x` from scope.

⇒ `print(x, end = e)` outputs `x` as a string followed by `e`; `end` is optional and defaults to a newline. `print(`x_1`, ..., `x_n`)` prints a tuple without parentheses or commas.

⇒ The `NoneType` has only one value, `None`. It's used as the return type of functions that only perform a side-effect, like printing to the screen. Use `type(None)` to refer to it.

Everything here works using Python3.

```
import sys
assert '3.8.1' == sys.version.split(' ')[0]
```

We'll use `assert y == f(x)` to show that the output of `f(x)` is `y`.

◇ Assertions are essentially "machine checked comments".

Explore built-in modules with `dir` and `help`

`dir(M)`	List of string names of all elements in module `M`
`help(M.f)`	Documentation string of function `f` in module `M`

```
import re
for member in sorted (dir(re)):
    if "find" in member:
        print (help ("re." + member))
```

⇒ Print alphabetically all regular expression utilities that mention `find`.

`help` can be called directly on a name; no need for quotes.

Arithmetic

Besides the usual operators `+, *, **, /, //, %, abs`, declare `from math import *` to obtain `sqrt, loq10, factorial, ...` —use `dir` to learn more, as mentioned above.

◇ Augmented assignments: `x ⊕= y` ≡ `x = x ⊕ y` for any operator ⊕.

◇ Floating point numbers are numbers with a decimal point.

◇ `**` for exponentiation and `%` for the remainder after division.

◇ `//`, floor division, discards the fractional part, whereas `/` keeps it.

◇ Numeric addition and *sequence* catenation are both denoted by `+`.

- ○ However: `1 + 'a'` ⇒ `error!`.

```
# Readability!
# '_' in numeric literals is ignored
assert 1000000 == 1_000_000

assert 1.2  == float("1.2")
assert -1 == int(float('-1.6'))
# float('a')
# ⇒ Crashes: 'a' is not a number
```

```
# Scientific notation: xey ≈ x * (10 ** y)
assert 250e-2 == 2.5 == 1 + 2 * 3 / 4.0

from math import *   # See below on imports
assert 2 == sqrt(4)
assert -inf < 123 < +inf
```

Conditionals

Booleans are a subtype (subclass) of integers, consisting of two values: `True` and `False`.

```
assert True == 1 and False == 0
assert issubclass(bool, int)
```

Consequently, we freely get Iverson brackets.

```
abs(x) ≈ x * (x > 0) - x * (x < 0)
```

Every "empty" collection is considered false! Non-empty values are truthy!

◇ `bool(x)` ⇒ Interpret object `x` as either true or false.

◇ E.g. 0, `None`, and empty tuples/lists/strings/dictionaries are falsey.

```
assert (False
        == bool(0)
        == bool("")
        == bool(None)
        == bool(())
        == bool([])
        == bool({}))
```

In Boolean contexts:

"x is empty"	≡	`not bool(x)`
`len(e) != 0`	≡	`bool(e)`
`bool(e)`	≡	`e`
`x != 0`	≡	`x`

User-defined types need to implement dunder methods `__bool__` or `__len__` .

Usual infix operations `and, or, not` for *control flow* whereas `&, |` are for Booleans only.

◇ `None or 4` ≈ `4` but `None | 4` crashes due to a type error.

`s`$_1$ `and ··· and s`$_n$	⇒	Do `s`$_n$ only if all `s`$_i$ "succeed"
`s`$_1$ `or ··· or s`$_n$	⇒	Do `s`$_n$ only if all `s`$_i$ "fail"

◇ `x = y or z` ⇒ assign `x` to be `y` if `y` is "non-empty" otherwise assign it `z`.

◇ Precedence: `A and not B or C` ≈ `(A and (not B)) or C`.

Value equality `==`, discrepancy `!=`; **Chained comparisons are conjunctive**; e.g.,

`x < y <= z`	≡	`x < y and y <= z`
`p == q == r`	≡	`p == q and q == r`

If-expressions must have an `else` clause, but *if-statements* need not `else` nor `elif` clauses; "`else if`" is invalid.

Expressions bind more tightly than statements; whence usually no need to parenthesise if-expressions.

```
# If-expression
expr₁ if condition else expr₂

# If-statement
if   condition₁: action₁
elif condition₂: action₂
elif condition₃: action₃
else           : default_action
```

Iterables

An *iterable* is an object which can return its members one at a time; this includes the (finite and ordered) *sequence types* —lists, strings, tuples— and non-sequence types —generators, sets, dictionaries. An iterable is any class implementing `__iter__` and `__next__`; an example is shown later.

- ◇ Zero-based indexing, `x[i]`, applies to sequence types only.
- ◇ We must have `-len(x) < i < len(x)` and `xs[-i]` ≈ `xs[len(x) - i]`.

We shall cover the general iterable interface, then cover lists, strings, tuples, etc.

Comprehensions provide a concise way to create iterables; they consist of brackets —() for generators, [] for lists, {} for sets and dictionaries— containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses.

```
(f(x) for x in xs if p(x))
```

⇒ A new iterable obtained by applying `f` to the elements of `xs` that satisfy `p` ⇐

E.g., the following prints a list of distinct pairs.

```
print ([(x, y) for x in [1,2,3] for y in (3,1,4) if x != y])
```

```
from itertools import count

evens = (2 * x for x in count())

# First 5 even naturals
for _, x in zip(range(5), evens):
    print (x)
```

Generators are "sequences whose elements are generated when needed"; i.e., are *lazy lists.*

If [,] are used in defining `evens`, the program will take forever to make a list out of the infinitly many even numbers!

Comprehensions are known as monadic do-notation in Haskell and Linq syntax in C#.

Generators are functions which act as a lazy streams of data: Once a `yield` is encountered, control-flow goes back to the caller and the function's state is persisted until another value is required.

```
# Infinite list of even numbers
def evens():
    i = 0;
    while True:
        yield i
        i += 2
```

```
xs = evens()
print (next (xs)) # ⇒ 0
print (next (xs)) # ⇒ 2
print (next (xs)) # ⇒ 4

# Print first 5 even numbers
for _, x in zip(range(5),evens()):
    print x
```

Notice that `evens` is just `count(0, 2)` from the itertools module.

Unpacking operation

- ◇ Iterables are "unpacked" with `*` and dictionaries are "unpacked" with `**`.
- ◇ Unpacking *syntactically* removes the outermost parenthesis ()/[]/{}.
- ◇ E.g., if `f` needs 3 arguments, then `f(*[x₁, x₂, x₃])` ≈ `f(x₁, x₂, x₃)`.
- ◇ E.g., printing a number of rows: `print(*rows, sep = '\n')`.
- ◇ E.g., coercing iterable `it`:

  `set(it)` ≈ `{*it}`, `list(it)` ≈ `[*it]`, `tuple(it)` ≈ `(*it,)`

Iterable unpacking syntax may also be used for assignments, where `*` yields lists.

```
x, *y, z = it   ≡    x = it[0]; z = it[-1]; y = list(it[1:len(it)-1])
                ⇒    [x] + ys + [z] = list(it)
```

E.g., `head , *tail = xs` to split a sequence.

In particular, since tuples only need parenthesis within expressions, we may write `x , y = e₁, e₂` thereby obtaining **simultaneous assignment**.
E.g., `x, y = y , x` to swap two values.

Loops let us *iterate over* iterables!

⇒ `break` exists a loop early; `continue` skips the current loop iteration.

⇒ Loops may be followed by an `else:` clause, which is executed *only* if the loop terminated by its condition failing —not due to a `break`!

```
# for-loop over a set
for x in {2, 3, 4}: print (x)
else: print ("for-loop is done")

# Looping over characters with indices
for i, x in enumerate('abc'):
    print (f"{i} goes to {x}")

# "while loop" over a tuple
i, xs = 0, (2, 3, 4)
while i < len(xs):
    print (xs[i])
    i += 1
```

Any user-defined class implementing `__iter__` and `__next__` can use loop syntax.

```
 for x in xs: f(x)
≈ it = iter(xs); while True: try: f(next(it)) except StopIteration: break
```

- ◇ `iter(x)` ⇒ Get an iterable for object `x`.
- ◇ `next(it)` ⇒ Get the current element and advance the iterable `it` to its next state.
  - ○ Raise StopIteration exception when there are no more elements.

Methods on Iterables

- ◇ `len` gives the length of (finite) iterables
  - ○ `len ((1, 2))` ⇒ 2; the extra parentheses make it clear we're giving *one tuple argument*, not *two integer arguments.*
- ◇ `x in xs` ⇒ check whether value `x` is a member of `xs`
  - ○ `x in y` ≡ `any(x == e for e in y)`, provided `y` is a finite iterable.
  - ○ `x in y` ≡ `y.__contains__(x)`, provided `y`'s class defines the method.
  - ○ `x not in y` ≡ `not x in y`

- `range(start, stop, step)` ⇒ An iterator of integers from `start` up to `stop-1`, skipping every other `step-1` number.
    - Associated forms: `range(stop)` and `range(start, stop)`.
- `reversed(xs)` returns a reversed iterator for `xs`; likewise `sorted(xs)`.
- `enumerate(xs)` ≈ `zip(xs, range(len(xs)))`
    - Pair elements with their indices.
- `zip`($\texttt{xs}_1$, ..., $\texttt{xs}_n$) is the iterator of tuples ($\texttt{x}_1$, ..., $\texttt{x}_n$) where $\texttt{x}_i$ is from $\texttt{xs}_i$.
    - Useful for looping over multiple iterables at the same time.
    - `zip(xs, ys)` ≈ `((x, y) for x in xs for y in ys)`
    - $\texttt{xs}_1$, ..., $\texttt{xs}_n$ = `zip`(*xs) ⇒ "unzip" xs, an iterable of tuples, into a tuple of (abstract) iterables $\texttt{xs}_i$, using the unpacking operation `*`.

```
xs , τ = [ {1,2} , [3, 4] ] , list
assert τ(map(tuple, xs)) == τ(zip(*(zip(*xs)))) == [(1,2) , (3,4)]
# I claim the first ‘‘==’’ above is true for any xs with:
assert len({len(x) for x in xs}) == 1
```

- `map`(f, $\texttt{xs}_1$, ..., $\texttt{x}_n$) is the iterable of values f $\texttt{x}_1$ ... $\texttt{x}_n$ where $\texttt{x}_i$ is from $\texttt{xs}_i$.
    - This is also known as *zip with f*, since it generalises the built-in `zip`.
    - `zip(xs, ys)` ≈ `map(lambda x, y: (x, y), xs, ys)`
    - `map(f, xs)` ≈ `(f(x) for x in xs)`
- `filter(p, xs)` ≈ `(x for x in xs if p(x))`
- `reduce`(⊕, [$\texttt{x}_0$, ..., $\texttt{x}_n$], e) ≈ e ⊕ $\texttt{x}_0$ ⊕ ⋯ ⊕ $\texttt{e}_n$; the initial value `e` may be omitted if the list is non-empty.

```
from functools import reduce

assert 'ABC' == reduce(lambda x, y: x + chr(ord(y) - 32), 'abc', '')
```

These are all instances of `reduce`:

- `sum`, `min`/`max`, `any`/`all` —remember "empty" values are falsey!

```
# Sum of first 10 evens
assert 90 == (sum(2*i for i in range(10)))
```

- Use `prod` from the `numpy` module for the product of elements in an iterable.

Flattening

Since, `sum(xs, e = 0)` ≈ `e + xs[0] + ⋯ + xs[len(xs)-1]` We can use `sum` as a generic "list of $\tau \to \tau$" operation by providing a value for `e`. E.g., lists of lists are catenated via:

```
assert [1, 2, 3, 4] == sum([[1], [2, 3], [4]], [])
assert (1, 2, 3, 4) == sum([(1,), (2, 3), (4,)], ())
# List of numbers where each number is repeated as many times as its value
assert [1, 2, 2, 3, 3, 3, 4, 4, 4, 4] == sum([i * [i] for i in range(5)], [])
```

Methods for sequences only

Numeric addition and *sequence* catenation are both denoted by `+`; however: `x + y` crashes when `type(x) != type(y)`.

```
assert 'hi' == 'h' + 'i'
assert (1, 2, 3, 4) == (1, 2) + (3, 4)
assert [1, 2, 3, 4] == [1, 2] + [3, 4]
```

Multiplication is iterated addition; not just for numbers, but for all *sequence types*!

```
assert "hi" * 2  == 2 * "hi" == "hihi"
assert (1,2) * 3 == (1, 2, 1, 2, 1, 2)
assert [1] * 3   == [1, 1, 1]
```

`xs.index(ys)` returns the first index in `xs` where `ys` occurs, or a `ValueError` if it's not present.

```
assert 1 == "abc".index('bc')
assert 0 == (1, 2, 3).index(1)
assert 1 == ['h', 'i'].index('i')
```

`xs.count(ys)` returns the number of times `ys` occurs as an element/substring of `xs`.

```
assert 1 == "abc".count('ab')
assert 0 == [1, 2, 3].count('ab')
assert 1 == [1, 2, 3].count(2)
assert 0 == [1, 2, 3].count([2, 3])
assert 1 == [1, [2, 3]].count([2, 3])
```

Sequences are Ordered

Sequences of the same type are compared lexicographically: Where `k = min(n, m)`, [$\texttt{x}_0$, ..., $\texttt{x}_n$] < [$\texttt{y}_0$, ..., $\texttt{y}_m$] ≡ $\texttt{x}_0$ < $\texttt{y}_0$ or ⋯ or $\texttt{x}_k$ < $\texttt{y}_k$ —recalling that Python's `or` is lazy; i.e., later arguments are checked only if earlier arguments fail to be true. Equality is component-wise.

```
assert [2, {}] != [3] # ⇒ Different lengths!
assert [2, {}] < [3]  # ⇒ True since 2 < 3.
assert (1, 'b', [2, {}]) < (1, 'b', [3])
```

$\boxed{\textbf{Tuples}}$

A *tuple* consists of a number of values separated by commas —parenthesis are only required when the tuples appear in complex expressions.

Tuples are immutable; no setters.

```
# Heterogeneous tuples
t = 1, 'b', [3], (4, 5)
```

But we can access then alter *mutable components* of a tuple; e.g., we can alter the list component of `t` ⇒

```
# Alter mutable component
t[2][0] = 33
assert t == (1, 'b', [33], (4, 5))
```

Getter is usual indexing, `xs[i]`.

```
empty_tuple = ()            # ⇒ ()
singleton_tuple = 'one',    # ⇒ ('one',)
# Note the trailing comma!
```

Convert `x` to a tuple with `tuple(x)`.

```
assert (('3', '4') == tuple(['3', '4'])
                   == tuple('34'))
```

Iverson brackets again!

```
(a, b)[c] ≈ a if c else b  # Eager!
```

Simultaneous assignment is really just tuple unpacking on the left and tuple packing on the right.

Strings

Strings are both `"`-enclosed and `'`-enclosed literals; the former easily allows us to include apostrophes, but otherwise they are the same.

- There is no separate character type; a character is simply a string of size one.
  - `assert 'hello' == 'he' + 'l' + 'lo' == 'he' 'l' 'lo'`
  - String literals separated by a space are automatically catenated.
- String characters can be accessed with [], but cannot be updated since strings are immutable. E.g., `assert 'i' == 'hi'[1]`.
- `str(x)` returns a (pretty-printed) string representation of an object.

String comprehensions are formed by joining all the strings in the resulting iterable —we may join using any separator, but the empty string is common.

```
assert '5 ≤ 25 ≤ 125' == (' ≤ '.join(str(5 ** i) for i in [1, 2, 3]))
```

- `s.join(xs).split(s)` ≈ `xs`
- `xs.split(s)` ⇒ split string `xs` into a list every time `s` is encountered

---

Useful string operations:

```
s.startswith(···)   s.endswith(···)
s.upper()           s.lower()
```

- `ord`/`chr` to convert between characters and integers.
- `input(x)` asks user for input with optional prompt `x`.
- E.g., `i = int(input("Enter int: "))` ⇒ gets an integer from user

---

f-strings are string literals that have an `f` before the starting quote and may contain curly braces surrounding expressions that should be replaced by their values.

```
name, age = "Abbas", 33.1
print(f"{name} is {age:.2f} years {'young' if age > 50 else 'old'}!")
# ⇒ Abbas is 33.10 years old!
```

F-strings are expressions that are evaluated at runtime, and are generally faster than traditional formatted strings —which Python also supports.

The brace syntax is `{expression:width.precision}`, only the first is mandatory and the last is either n`f` or n`e` to denote n-many decimal points or scientific notation, respectively.

Lists

Python supports zero-indexed heterogeneous lists.

Like all sequence types, we access values with indices `xs[0]` and modify them in the same way. Above `xs[12]` yields an out of range error.

```
# Making lists
xs = []
xs.append(1)
xs.append([2, 'a'])
xs.append('b')
# or:
xs = [1, [2, 'a'], 'b']
```

Besides all of the iterable methods above, for lists we have:

- `list(cs)` ⇒ turns a string/tuple into the list of its characters/components
- `xs.remove(x)` ⇒ remove the first item from the list whose value is `x`.
- `xs.index(x)` ⇒ get first index where `x` occurs, or error if it's not there.
- `xs.pop(i)` ≈ `(x := xs[i], xs := xs[:i] + xs[i+1:])[0]`
  - are covered below; if `i` is omitted, it defaults to `len(xs)-1`.
  - Lists are thus stacks with interface `append`/`pop`.
- For a list-like container with fast appends and pops on either end, see the deque collection type.

Sets

`set(xs)` to transform a sequence into a set, which is a list without repetitions.

Useful methods `a.`m`(b)` where m is `intersection`, `union`, `difference`, `symmetric_difference`.

```
# Two ways to make sets; no duplicates!
assert {1, 2, 3} == set([3, 1, 1, 2])

# Set comprehension
{x for x in 'abracadabra'
   if  x not in 'abc'}
# ⇒ {'d', 'r'}
```

Dictionaries

Note that `{}` denotes the empty dictionary, not the empty set.

A *dictionary* is like a list but indexed by user-chosen *keys*, which are members of any immutable type. *It's really a set of "key:value" pairs.*

E.g., a dictionary of numbers along with their squares can be written explicitly (below left) or using a comprehension (below right).

```
assert {2: 4, 4: 16, 6: 36} == {x: x**2 for x in (2, 4, 6)}
```

Hierarchical Tree Structures

```
you = { "kid1": "Alice"
      , "kid2": { "kid1": "Bobert"
                , "kid2": "Mary"
                }
      }
```

"Case Statements"

```
i, default = 'k' , "Dec"
x = { 'a': "Jan"
    , 'k': "Feb"
    , 'p': "Mar"
    }.get(i, default)
assert x == 'Feb'
```

Alternatively: Start with `you = {}` then later add key-value pairs: `you[key] = value`.

```
assert 'Bobert' == you["child2"]['child1'] # access via indices
del you['child2']['child2']               # Remove a key and its value
assert 'Mary' not in you['child2'].values()
```

- `list(d)` ⇒ list of keys in dictionary `d`.
- `d.keys()`, `d.values()` ⇒ get an iterable of the keys or the values.
- `k in d` ⇒ Check if key `k` is in dictionary `d`.
- `del d[k]` ⇒ Remove the key-value pair at key `k` from dictionary `d`.
- `d[k] = v` ⇒ Add a new key-value pair to `d`, or update the value at key `k` if there is one.
- `dict(xs)` ⇒ Get a dictionary from a list of key-value tuples.

  When the keys are strings, we can specify pairs using keyword arguments: `dict(me = 12, you = 41, them = 98)`.

Conversely, `d.items()` gives a list of key-value pairs; which is useful to have when looping over dictionaries.

In dictionary literals, later values will always override earlier ones:

```
assert dict(x = 2) == {'x':1, 'x':2}
```

Dictionary update: `d = {**d, key`$_1$`:value`$_1$`, ..., key`$_n$`:value`$_n$`}`.

Splicing

`xs[start:stop:step]` ≈ the subsequence of `xs` from `start` to `stop-1` skipping every `step-1` element. All are optional, with `start`, `stop`, and `step` defaulting to `0`, `len(xs)`, and `1`; respectively.

- ◇ The start is always included and the end always excluded.
- ◇ `start` may be negative: $-n$ means the n-th item from the end.
- ◇ All slice operations return a new sequence containing the requested elements.
- ◇ One colon variant: `xs[start:stop]`, both `start` and `stop` being optional.
- ◇ *Slicing applies to sequence types only* —i.e., types implementing `__getitem__`.

```
xs = [11, 22, 33, 44, 55]

assert xs[3:-5]  == []
assert xs[3:7]   == [44, 55]
assert (   xs[3:77]
        == xs[3: min(77, len(xs))]
        == xs[3:5])
```

```
assert "ola"     == "hola"[1:]
assert (3, 2, 1) == (1, 2, 3)[::-1]

assert xs[-1::] == [55]

n, N = 10, len(xs)
assert xs[-n::] == xs[max(0, N - n)::]
```

Useful functions via splicing

`xs[:n]`	⇒	take first n items
`xs[0]`	⇒	head of `xs`
`xs[n:]`	⇒	drop first n items
`xs[1:]`	⇒	tail of `xs`
`xs[-1]`	⇒	last element of `xs`
`xs[::k]`	⇒	get every k-th value
`n * [x]`	⇒	the list consisting of `x` repeated n-times

Splice laws

`xs[:]`	≈	`xs`
`xs[::]`	≈	`xs`
`xs[0:len(xs)]`	≈	`xs`
`xs[::-1]`	≈	`reversed(xs)`
`xs[:n] + xs[n:]`	≈	`xs`
`len(xs[m:n])`	≈	`n - m`

```
  xs[m:n] = ys
≡ xs = xs[:m] + ys + xs[n:]
```

Assignment to slices is possible, resulting in sequences with possibly different sizes.

```
xs      = list(range(10)) # ⇒  xs ≈ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
xs[3:7] = ['a', 'b']      # ⇒  xs ≈ [0, 1, 2, 'a', 'b',   7, 8, 9]
```

Other operations via splicing:

- ◇ `0 == s.find(s[::-1])` ⇒ string `s` is a palindrome
- ◇ `inits xs` ≈ `[xs[0:i] for i in range(1 + len(xs))]`
- ◇ `segs xs` ≈ `[xs[i:j] for i in range(len(xs)) for j in range(i, len(xs))]`

Functions

Functions are first-class citizens: Python has one namespace for functions and variables, and so there is no special syntax to pass functions around or to use them anywhere, such as storing them in lists.

- ◇ `return` clauses are optional; if there are none, a function returns `None`.
- ◇ Function application always requires parentheses, even when there are no arguments.
- ◇ Any object `x` can be treated like a function, and use the `x(···)` application syntax, if it implements the `__call__` method: `x(···)` ≈ `x.__call__(···)`. The `callable` predicate indicates whether an object is callable or not.
- ◇ Functions, and classes, can be nested without any special syntax; the nested functions are just new local values that happen to be functions. Indeed, nested functions can be done with `def` or with assignment and `lambda`.
- ◇ Functions can receive a variable number of arguments using `*`.

```
def compose(*fs):
    """Given many functions f₀,...,fₙ return a new one: λ x. f₀(···(fₙ(x))···)"""
    def seq(x):
        seq.parts = [f.__name__ for f in fs]
        for f in reversed(fs):
            x = f(x)
        return x
    return seq

print (help(compose))  # ⇒ Shows the docstring with function type
compose.__doc__ = "Dynamically changing docstrings!"

# Apply the “compose” function;
# first define two argument functions in two ways.

g = lambda x: x + 1
def f(x): print(x)

h = compose(f, g, int)
h('3')                          # ⇒ Prints 4
print(h.parts)                  # ⇒ ['f', '<lambda>', 'int']
print (h.__code__.co_argcount) # ⇒ 1; h takes 1 argument!

# Redefine “f” from being a function to being an integer.
f = 3
# f(1) # ⇒ Error: “f” is not a function anymore!
```

Note that `compose()` is just the identity function `lambda x: x`.

The first statement of a function body can optionally be a ‘docstring’, a string enclosed in three double quotes. You can easily query such documentation with `help(functionName)`. In particular, `f.__code__.co_argcount` to obtain the number of arguments `f` accepts.

That functions have attributes —state that could alter their behaviour— is not at all unexpected: Functions are objects; Python objects have attributes like `__doc__` and can have arbitrary attributes (dynamically) attached to them.

A `lambda` is a single line expression; you are prohibited from writing statements like `return`, but the semantics is to do the `return`.

`lambda args:` (x_0 := e_0, ..., x_n := e_n)[k] is a way to perform n-many stateful operations and return the value of the k-th one. See `pop` above for lists; are covered below.

For fresh name `x`, a **let-clause** *"let $x = e$ in $\cdots$"* can be simulated with `x = e; ...; del x`. However, in combination with , lambda's ensure a new local name: `(lambda x = e: ···)()`.

Default & keyword argument values are possible

```
def go(a, b=1, c='two'):
    """Required 'a', optional 'b' and 'c'"""
    print(a, b, c)
```

Keyword arguments must follow positional arguments; order of keyword arguments (even required ones) is not important.

- ◇ Keywords cannot be repeated.

```
go('a')          # ⇒ a 1 two ;; only required, positional
go(a='a')        # ⇒ a 1 two ;; only required, keyword
go('a', c='c')   # ⇒ a 1 c   ;; out of order, keyword based
go('a', 'b')     # ⇒ a b two ;; positional based
go(c='c', a='a') # ⇒ a 1 c   ;; very out of order
```

Dictionary arguments

After the *required* positional arguments, we can have an arbitrary number of optional/positional arguments (a tuple) with the syntax `*args`, after that we may have an arbitrary number of *optional* keyword-arguments (a dictionary) with the syntax `**args`.

The reverse situation is when arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. Recall, from above, that we do so using the `*` operator; likewise `**` is used to unpack dictionaries.

- ◇ E.g., if `f` needs 3 arguments, then `f(*[`x_1`, `x_2`, `x_3`])` ≈ `f(`x_1`, `x_2`, `x_3`)`.

```
def go(a, *more, this='∞', **kwds):
    print (a)
    for m in more: print(m)
    print (this)
    for k in kwds: print(f'{k} ↦ {kwds[k]}')
    return kwds['neato'] if 'neato' in kwds else -1

# Elementary usage
go(0)                                # ⇒ 0 ∞
go(0, 1, 2, 3)                       # ⇒ 0 1 2 3 ∞
go(0, 1, 2, this = 8, three = 3) # ⇒ 0 1 2 8 three ↦ 3
go(0, 1, 2, three=3, four = 4)   # ⇒ 0 1 2 ∞ three ↦ 3 four ↦ 4

# Using "**"
args = {'three': 3, 'four': 4}
go(0, 1, 2, **args) # ⇒ 0 1 2 ∞ three ↦ 3 four ↦ 4
```

```
# Making use of a return value
assert 5 == go (0, neato = 5)
```

Type Annotations

We can annotate functions by expressions —these are essentially useful comments, and not enforced at all— e.g., to provide type hints. They're useful to document to human readers the intended types, or used by third-party tools.

```
# A function taking two ints and returning a bool
def f(x:int, y : str = 'neat') -> bool:
    return str(x) # Clearly not enforced!

print (f('hi')) # ⇒ hi; Typing clearly not enforced

print(f.__annotations__) # ⇒ Dictionary of annotations
```

Currying: Fixing some arguments ahead of time.

```
  partial(f, v₀, ..., vₙ)
≈ lambda x₀, ..., xₘ:
    f(v₀, ..., vₙ, x₀, ..., xₘ)
```

```
from functools import partial

multiply  = lambda x, y, z: z * y + x
twice     = partial(multiply, 0, 2)
assert 10 == twice(5)
```

Using decorators and classes, we can make an 'improved' partial application mechanism —see the final page.

Decorators allow us to modify functions in orthogonal ways, such as printing values when debugging, without messing with the core logic of the function. E.g., to do pre-processing before and after a function call —e.g., `typed` below for this standard template. Decorators are just functions that alter functions, and so they can return anything such as an integer thereby transforming a function name into an integer variable.

```
@decorator_2
@decorator_1
def fun(args):
    ...
```

≈

```
def fun(args):
    ...
fun = decorator_2(decorator_1(fun))
```

The decoration syntax `@ d f` is a convenient syntax that emphasises code acting on code.

Decorators can be helpful for functions we did not write, but we wish to advise their behaviour; e.g., `math.factorial = my_decorator(math.factorial)` to make the standard library's `factorial` work in new ways.

When decorating, we may use *args and **kwargs in the inner wrapper function so that it will accept an arbitrary number of positional and keyword arguments. See `typed` below, whose inner function accepts any number of arguments and passes them on to the function it decorates.

We can also use decorators to add a bit of type checking at runtime:

```python
import functools

# “typed” makes decorators; “typed(s_1, ..., s_n, t)” is an actual decorator.
def typed(*types):
    *inTys, outT = types
    def decorator(fun):
        @functools.wraps(fun)
        def new(*args, **kwdargs):
            # (1) Preprocessing stage
            if any(type(w := arg) != ty for (arg, ty) in zip(args, inTys)):
                nom = fun.__name__
                raise TypeError (f"{nom}: Wrong input type for {w!r}.")
            # (2) Call original function
            result = fun(*args, **kwdargs) # Not checking keyword args
            # (3) Postprocessing stage
            if type(result) != outT:
                raise TypeError ("Wrong output type!")
            return result
        return new
    return decorator
```

After being decorated, function attributes such as `__name__` and `__doc__` refer to the decorator's resulting function. In order to have it's attributes preserved, we copy them over using `@functools.wraps` decorator —or by declaring `functools.update_wrapper(newFun, oldFun)`.

```python
# doit : str × list × bool → NoneType
@typed(str, list, bool, type(None))
def doit(x, y, z = False, *more):
    print ((ord(x) + sum(y)) * z, *more)
```

Notice we only typecheck as many positions as given, and the output; other arguments are not typechecked.

```python
# ⇒ TypeError: doit: Wrong input type for 'bye'!
doit('a', [1, 2], 'bye')

# ⇒ 100 n i ;; typechecking succeeds
doit('a', [1, 2], True, 'n', 'i')

# ⇒ 0; Works with defaults too ;-)
doit(x, y)

# ⇒ 194; Backdoor: No typechecking on keyword arguments!
doit('a', z = 2, y = {})
```

The implementation above matches the `typed` specification, but the one below does not and so always crashes.

```python
# This always crashes since
# the result is not a string.
@typed(int, str)
def always_crashes(x):
    return 2 + x
```

Note that `typed` could instead *enforce* type annotations, as shown before, at run time ;-)

An easier way to define a family of decorators is to define a decorator-making-decorator!

Object-Oriented Programming

Classes bundle up data and functions into a single entity; *Objects* are just values, or *instances*, of class types. That is, a class is a record-type and an object is a tuple value.

1. A Python *object* is just like a real-world object: It's an entity which has attributes —*a thing which has features.*
2. We *classify* objects according to the features they share.
3. A *class* specifies properties and behaviour, an implementation of which is called an *object*. Think class is a cookie cutter, and an actual cookie is an object.
4. Classes are also known as “bundled up data”, structures, and records.

   They let us treat a collection of data, including methods, as one semantic entity. E.g., rather than speak of name-age-address tuples, we might call them person objects.

   Rather than “acting on” tuples of data, an object “knows how to act”; we shift from doit(x_1, ..., x_n) to x.doit(). We *abstract away* the n-many details into 1 self-contained idea.

What can we learn from an empty class?

```python
class Person:
    """An example, empty, class."""
    pass
```

`pass` is the “do nothing” statement. It's useful for writing empty functions/classes that will be filled in later or for explicitly indicating do-nothing cases in complex conditionals.

```python
# We defined a new type!
assert isinstance(Person, type)

# View information of the class
print (help(Person))
# Or use: Person.__name__,
# Person.__doc__, Person.__dict__

# Let's make a Person object
jasim = Person()
assert Person == type(jasim)
assert isinstance(jasim, Person)
```

Instance (reference) equality is compared with `is`.

`x is y ≡ id(x) == id(y)`

`id(x)` is a unique number identifying object `x`; usually its address in memory.

```python
jason = jasim
qasim = Person ()
assert jason is jasim and jasim is jason
assert qasim is not jasim
```

```python
# Check attributes exist before use
assert not hasattr(jasim, 'work')

# Dynamically add new (instance) attributes
jasim.work = 'farmer'
jasim.nick = 'jay'

# Delete a property
del jasim.nick

# View all attribute-values of an object
print(jasim.__dict__) # {'work': 'farmer'}
```

Look at that, classes are just fancy dictionary types!

The converse is also true: `class X: a = 1` ≈ `X = type('X', (object,), dict(a = 1))`

Let's add more features!

`# [0]` An `__init__` method is called whenever a new object is created via `Person(name, age)`. It *constructs* the object by *initialising* its necessary features.

`# [1]` The argument `self` refers to the object instance being created and `self.x = y` is the creation of an attribute `x` with value `y` for the newly created object instance. Compare `self` with `jasim` above and `self.work` with `jasim.work`. It is convention to use the name `self` to refer to the current instance, you can use whatever you want but it must be the first argument.

`# [2]` Each `Person` instance has their own `name` and `work` features, but they universally share the `Person.__world` feature. Attributes starting with two underscores are *private*; they can only be altered within the definition of the class. Names starting with no underscores are *public* and can be accessed and altered using dot-notation. Names starting with one underscore are *protected*; they can only be used and altered by children classes.

```
class Person:
    __world = 0  # [2]

    def __init__(self, name, work): # [0]
        self.name = name
        self.work = work
        Person.__world += 1

    def speak(me): # [1] Note, not using ‘‘self’’
        print (f"I, {me.name}, have the world at my feet!")

    # Implementing __str__ allows our class to be coerced as string
    # and, in particular, to be printed.
    def __str__(self):
        return (f"In a world of {Person.__world} people, "
                f"{self.name} works at {self.work}")

    # [3] Any class implementing methods __eq__ or __lt__
    # can use syntactic sugar == or <, respectively.
    def __eq__(self, other):
        return self.work == other.work

    # We can loop over this class by defining __iter__,
    # to setup iteration, and __next__ to obtain subsequent elements.
    def __iter__(self):
        self.x = -1
        return self

    def __next__(self):
        self.x += 1
        if self.x < len(self.name): return self.name[self.x]
        else: raise StopIteration
```

Making People

```
jason = Person('Jasim', "the old farm")
kathy = Person('Kalthum', "Jasim's farm")
print(kathy) # ⇒ In a world of 2 people, Kalthum works at Jasim's farm
# Two ways to use instance methods
jason.speak()       # ⇒ I, Jasim, have the world at my feet!
Person.speak(jason)
```

The following code *creates a new public feature that happens to have the same name as the private one*. This has no influence on the private feature of the same name! See `# [2]` above.

```
Person.__world = -10
# Check that our world still has two people:
print(jason) # ⇒ In a world of 2 people, Jasim works at the old farm
```

Syntax Overloading: Dunder Methods

`# [3]` Even though `jasim` and `kathy` are distinct people, in a dystopian world where people are unique up to contribution, they are considered “the same”.

```
kathy.work = "the old farm"
assert jason is not kathy
assert jason == kathy
```

We can use any Python syntactic construct for new types by implementing the dunder —“d”ouble “under”score— methods that they invoke. This way new types become indistinguishable from built-in types. E.g., implementing `__call__` makes an object behave like a function whereas implementing `__iter__` and `__next__` make it iterable —possibly also implementing `__getitem__` to use the slicing syntax `obj[start:stop]` to get a ‘sub-segment’ of an instance. Implementing `__eq__` and `__lt__` lets us use `==`, `<` which are enough to get `<=`, `>` if we decorate the class by the `@functools.total_ordering` decorator. Reflected operators `__r`*op*`__` are used for arguments of different types:
`x ⊕ y` ≈ `y.__r⊕__(x)` if `x.__⊕__(y)` is not implemented.

```
# Loop over the ‘‘jason’’ object; which just loops over the name’s letters.
for e in jason:
    print (e)  # ⇒ J \n a \n s \n i \n m

# Other iterable methods all apply.
print(list(enumerate(jason)) # ⇒ [(0, ’J’), (1, ’a’), (2, ’s’), ...]
```

One should not have attributes named such as `__attribute__`; the dunder naming convention is for the Python implementation team.

- ◇ Here is a list of possible dunder methods.
- ◇ `__add__` so we can use `+` to merge instances —then use `sum` to ‘add’ a list of elements.
- ◇ Note: *h*`(x)` ≈ `x.__`*h*`__` for *h*: `len`, `iter`, `next`, `bool`, `str`.

Extension Methods

```
# ‘‘speak’’ is a public name, so we can assign to it:
# (1) Alter it for ‘‘jason’’ only
jason.speak =  lambda: print(f"{jason.name}: Hola!")
# (2) Alter it for ALL Person instances
Person.speak = lambda p: print(f"{p.name}: Salam!")
jason.speak() # ⇒ Jasim: Hola!
kathy.speak() # ⇒ Kalthum: Salam!
```

Notice how `speak()` above was altered. In general, we can “mix-in new methods” either at the class level or at the instance level in the same way.

```
# New complex method
def speak(self):
    ···

# Add it at the class level
Person.speak = speak

# Remove ‘‘speak’’ from
# the current scope
del speak
```

This ability to extend classes with new functions does not work with the builtin types like `str` and `int`; neither at the class level nor at the instance level. If we want to inject functionality, we can simply make an empty class like the first incarnation of `Person` above. An example, `PartiallyAppliedFunction`, for altering how function calls work is shown on the right column ⇒

Inheritance

A class may *inherit* the features of another class; this is essentially automatic copy-pasting of code. This gives rise to *polymorphism*, the ability to “use the same function on different objects”: If class `A` has method `f()`, and classes `B` and `C` are children of `A`, then we can call `f` on `B`- and on `C`-instances; moreover `B` and `C` might redefine `f`, thereby ‘overloading’ the name, to specialise it further.

```
class Teacher(Person):
    # Overriding the inherited __init__ method.
    def __init__(self, name, subject):
        super().__init__(name, f'the university teaching {subject}')
        self.subject = subject

assert isinstance(Teacher, type)
assert issubclass(Teacher, Person)
assert issubclass(Person, object)
# The greatest-grandparent of all classes is called ‘‘object’’.

moe = Teacher('Ali', 'Logic')
assert isinstance(moe, Teacher) # By construction.
assert isinstance(moe, Person)  # By inheritance.
print(moe)
# ⇒ In a world of 3 people, Ali works at the university teaching Logic
```

Decorators and Classes

Since `@C f` stands for `f = C(f)`, we can decorate via *classes* `C` whose `__init__` method takes a function. Then `@C f` will be a class! If the class implements `__call__` then we can continue to treat `@C f` as if it were a (stateful) function.

In turn, we can also decorate class methods in the usual way. E.g., when a method x`(self)` is decorated `@property` , we may attach logic to its setter `obj.`x `= ···` and to its getter `obj.`x!

We can decorate an entire class `C` as usual; `@dec C` still behaves as update via function application: `C = dec(C)`. This is one way to change the definition of a class dynamically.

- ⋄ E.g., to implement design patterns like the singleton pattern.

A class decorator is a function from classes to classes; if we apply a function decorator, then only the class’ constructor is decorated —which makes sense, since the constructor and class share the same name.

Example: Currying via Class Decoration

Goal: We want to apply functions in many ways, such as `f(`x_1`, ..., `x_n`)` and `f(`x_1`, ..., `x_i`)(`x_{i+1}`, ..., `x_n`)`; i.e., all the calls on the right below are equivalent.

```
@curry
def doit(x, y, z):
    print('got', x, y, z)
```

```
doit(1)(2)(3)
doit(1, 2)(3)
doit(1)(2, 3)
doit(1, 2, 3)
doit(1, 2, 3, 666, '∞') # Ignore extra args
```

The simplest thing to do is to transform `f = lambda `x_1`, ..., `x_n`: body` into `nestLambdas(f, [], f.__code__.co_argcount) = lambda `x_1`: ...: lambda `x_n`: body`.

```
def nestLambdas (func, args, remaining):
   if remaining == 0: return func(*args)
   else: return lambda x: nestLambdas(func, args + [x], remaining - 1)
```

However, the calls shift from `f(`v_1`, ..., `v_k`)` to `f(`v_1`)(`v_2`)···(`v_k`)`; so we need to change what it means to call a function. As already mentioned, we cannot extend built-in classes, so we’ll make a wrapper to slightly alter what it means to call a function on a smaller than necessary amount of arguments.

```
class PartiallyAppliedFunction():

    def __init__(self, func):
        self.value = nestLambdas(func, [], func.__code__.co_argcount)

    def __mul__ (self, other):
        return PartiallyAppliedFunction(lambda x: self(other(x)))

    apply = lambda self, other: other * self if callable(other) else self(other)
    def __rshift__(self, other):  return self.apply(other)
    def __rrshift__(self, other): return self.apply(other)

    def __call__(self, *args):
        value = self.value
        for a in args:
            if callable(value):
                value = value(a)
        return PartiallyAppliedFunction(value) if (callable(value)) else value

curry = PartiallyAppliedFunction   # Shorter convenience name
```

The above invocation styles, for `doit`, now all work ^_^

Multiplication now denotes function composition and the (‘r’eflected) ‘r’ight-shift denotes forward-composition/application:

`(g * f(`v_1`, ..., `v_m`))(`x_1`, ..., `x_n`) = g(f (`v_1`, ..., `v_m`, `x_1`))(`x_2`, ..., `x_n`)`

```
@curry
def f(x, y, z): return x + y + z

@curry
def g(x, y): return [x] * y
```

```
assert(   (g * f(3, 1))(9, 4)
       == (f(3, 1) >> g)(9, 4)
       == [13, 13, 13, 13])

assert (   ['a', 'a', 'b']
        == 2 >> g('a')
           >> curry(lambda x: x + ['b']))
```

Named Expressions

The value of a "walrus" expression `x := e` is the value of `e`, but it also introduces the name `x` into scope. The name `x` must be an atomic identifier; e.g., not an unpacked pattern or indexing; moreover `x` cannot be a `for`-bound name.

"subexpression reuse"

```
     f(e, e)
  ≈ f(x := e, x)
```

"if-let"

```
  x = e; if p(x): f(x)
≈ if p(x := e): f(x)
```

This can be useful to capture the value of a *truthy* item so as to use it in the body:

```
  if e: x = e; f(x)
≈ if (x := e): f(x)
```

"while-let"

```
  while True:
      x = input(); if p(x): break; f(x)
≈
  while p(x := input()): f(x)
```

"witness/counterexample capture"

```
if any(p(witness := x) for x in xs):
    print(f"{witness} satisfies p")

if not all(p(witness := x) for x in xs):
    print(f"{witness} falsifies p")
```

"Stateful Comprehensions"

```
  partial sums of xs
≈ [sum(xs[: i + 1]) for i in range(len(xs))]
≈ (total := 0, [total := total + x for x in xs])[1]
```

Walrus introduces new names, what if we wanted to check if a name already exists?

```
# alter x if it's defined else, use 7 as default value
x = x + 2 if 'x' in vars() else 7
```

Modules

⇒ Each Python file `myfile.py` determines a module whose contents can be used in other files, which declare `import myfile`, in the form `myfile.component`.

⇒ To use a function `f` from `myfile` without qualifying it each time, we may use the `from` import declaration: `from myfile import f`.

⇒ Moreover, `from myfile import *` brings into scope all contents of `myfile` and so no qualification is necessary.

⇒ To use a smaller qualifier, or have conditional imports that alias the imported modules with the same qualifier, we may use `import thefile as newNameHere`.

⇒ A Python package is a directory of files —i.e., Python modules— with a (possibly empty) file named `__init__.py` to declare the directory as a package.

⇒ If `P` is a package and `M` is a module in it, then we can use `import P.M` or `from P import M`, with the same behaviour as for modules. The init file can mark some modules as private and not for use by other packages.

Reads

- Dan Bader's Python Tutorials —bite-sized lessons
  - Likewise: w3schools Python Tutorial
- www.learnpython.org —an interactive and tremendously accessible tutorial
- The Python Tutorial —really good introduction from python.org
- `https://realpython.com/` —real-world Python tutorials
- Python for Lisp Programmers
- A gallery of interesting Jupyter Notebooks —interactive, 'live', Python tutorials
- How to think like a computer scientist —Python tutorial that covers turtle graphics as well as drag-and-drop interactive coding and interactive quizzes along the way to check your understanding; there are also videos too!
- Monads in Python —Colourful Python tutorials converted from Haskell
- Teach Yourself Programming in Ten Years