Python 3, An Illustrated Tour

@__mharrison__

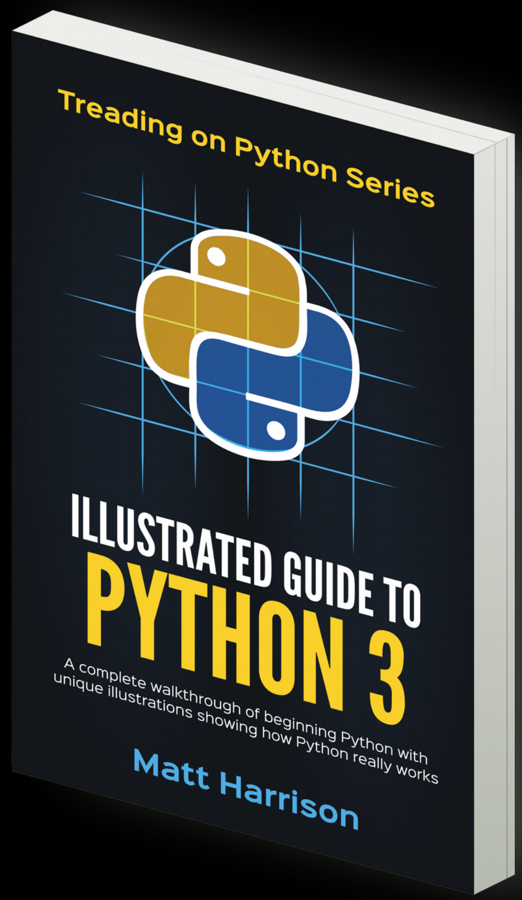
METASNAKE

Introduction

Introduction

Changes & new features in Python 3

About Matt



- Author of *Illustrated Guide to Python*, *Guide to Learning Pandas*, *Tiny Python 3.6 Notebook*
- Consultant and Trainer for MetaSnake
- Ran Utah Python for 5 years
- Python since 2000
- `@__mharrison__`

Who is this for?

- Old Python Programmers
- New Python Programmers
- Looking to leverage latest features

Content

Virtual Environments

Classes

Syntax

Unicode

Fstrings

Functions

Asyncio

Annotation Tools

Numbers

Std Library

Annotations

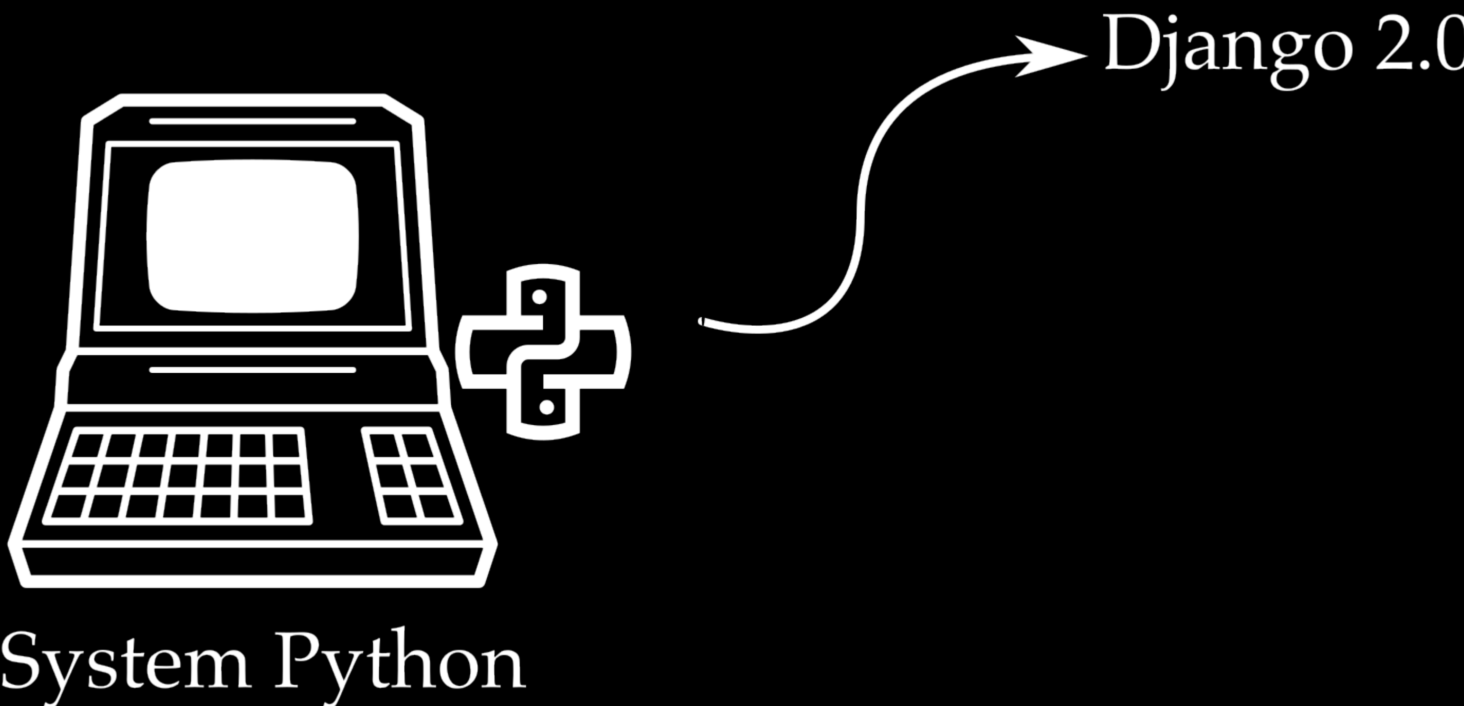
Unpacking

Virtual environments
(PEP 405 - 3.3)
PIP (PEP 453 - 3.4)

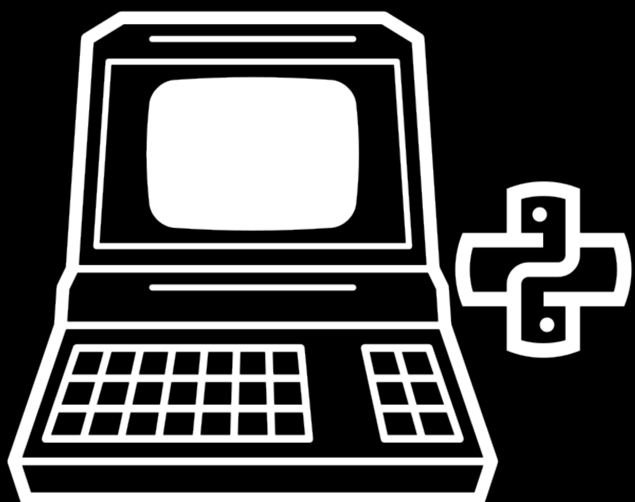
Virtual Environments

- Per project dependencies
- Easy to install/upgrade deps

Virtual Environments



Virtual Environments

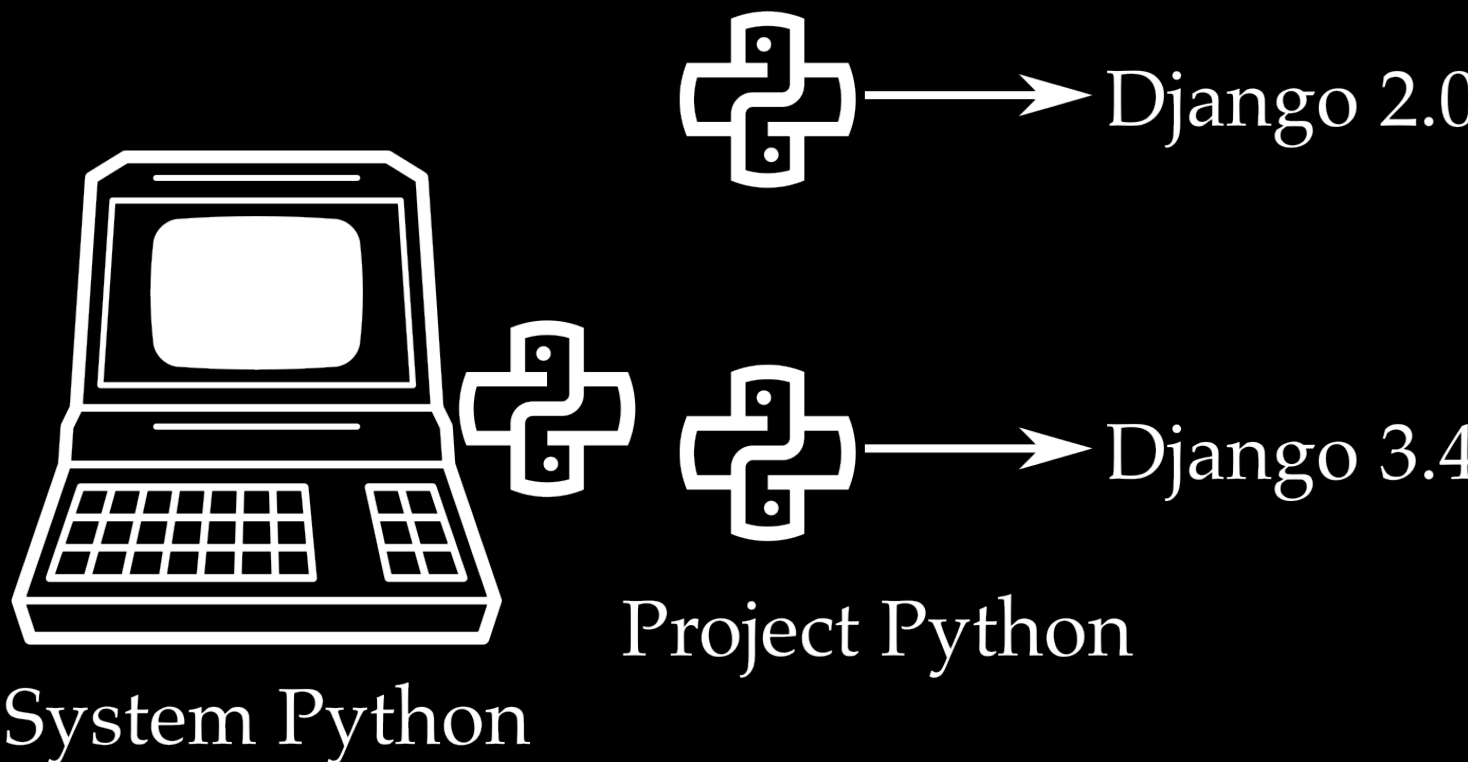


Django 2.0

Django 3.4

System Python

Virtual Environments



Create a Virtual Environment

Unix:

```
$ python3 -m venv /path/to/env
$ source /path/to/env/bin/activate
(env) $
```

Create a Virtual Environment

Windows:

```
c:\>c:\Python36\python -m venv c:\path\to\env
c:\>c:\path\to\env\Scripts\activate.bat
(env) c:\>
```

`-m` Switch

Executes a module. Make sure we know what Python we are using to create our virtualenv

What to do inside virtualenv?

(Un)Install using `pip`

- `pip install foo` - Install package foo
- `pip install -r req.txt` - Install requirements file
- `pip install -e` - Install package in "edit" mode
- `pip freeze` - Output install packages
- `pip uninstall foo` - Uninstall foo

Warning

Make sure you run `pip` inside a virtual environment or it may not install where you think it is.

Pipenv

Recommended tool for managing deps

https://packaging.python.org/tutorials/managing-dependencies/#managing-dependencies

Install

Install in *user installation* (user isolation):

```
$ python3 -m pip install --user pipenv
```

Install

Make sure `PATH` has user install path. (`pipenv` can't run as a module)

```
$ python3 -m site --user-base
```

Add `bin` or `Python36\Scripts\` to `PATH`.

Install

Unix system:

```
$ head ~/.bash_profile
export PATH="$HOME/Library/Python/3.6/bin:$PATH"

$ source ~/.bash_profile
```

Install

Windows instructions:

- Launch environment editor (type `env` in Search Box)
- Edit environment to add userbase + `Python36\Scripts\`
- Relaunch Command Prompt (type `cmd` in Search Box)

https://msdn.microsoft.com/en-us/library/windows/desktop/bb776899(v=vs.85).aspx

Using Pipenv

Make a directory:

```
$ mkdir blockchain
$ cd blockchain
$ pipenv install py.test
```

You should have the following:

```
$ pwd
blockchain
$ tree .
.
├── Pipfile
└── Pipfile.lock
```

Pipfile

- Supersedes `requirements.txt` file
- One file (`Pipfile`) to support dev, default (production)
- `Pipfile.lock` stores installation details (versions, hashes)

Using Pipenv

On my system virtual env is in (will use `.venv` if `PIPENV_VENV_IN_PROJECT` environment variable exists):

```
/Users/matt/.local/share/virtualenvs/blockchain-aWoplWKu/
```

Using Pipenv

```
$ pipenv run python  # just run python
$ pipenv shell       # activate
```

More Pipenv

```
$ pipenv --venv     # location of env
$ pipenv --py       # location of python
$ pipenv install pkg --dev  # dev dependency
$ pipenv graph      # dependency graph
$ pipenv lock       # create lockfile
$ pipenv uninstall pkg      # remove pkg
```

Assignment

venv_test.py

Unicode

Terms

- Character - Single letter
- Glyph - Visual representation of a character
- Code point - Unicode numeric (hex) description of character
- Encoding - Mapping of byte stream to code point (may be more than one byte)

Example

- Character - Omega
- Glyph - Ω
- Code point - U+2126
- Encoding - `b'\xe2\x84\xa6'` (UTF-8) `b'\xff\xfe&!'` (UTF-16)

Creating Strings

Can use glyph, codepoint, name, or `chr` integer:

```
>>> o1 = 'Ω'          # glyph
>>> o2 = '\u2126'     # codepoint (hex 8486)
>>> o3 = '\N{OHM SIGN}'   # name
>>> o4 = chr(8486) # character from ordinal
>>> o1
'Ω'
>>> o1 == o2 == o3 == o4
True
```

Getting Name

```
>>> import unicodedata
>>> unicodedata.name(o1)
'OHM SIGN'
```

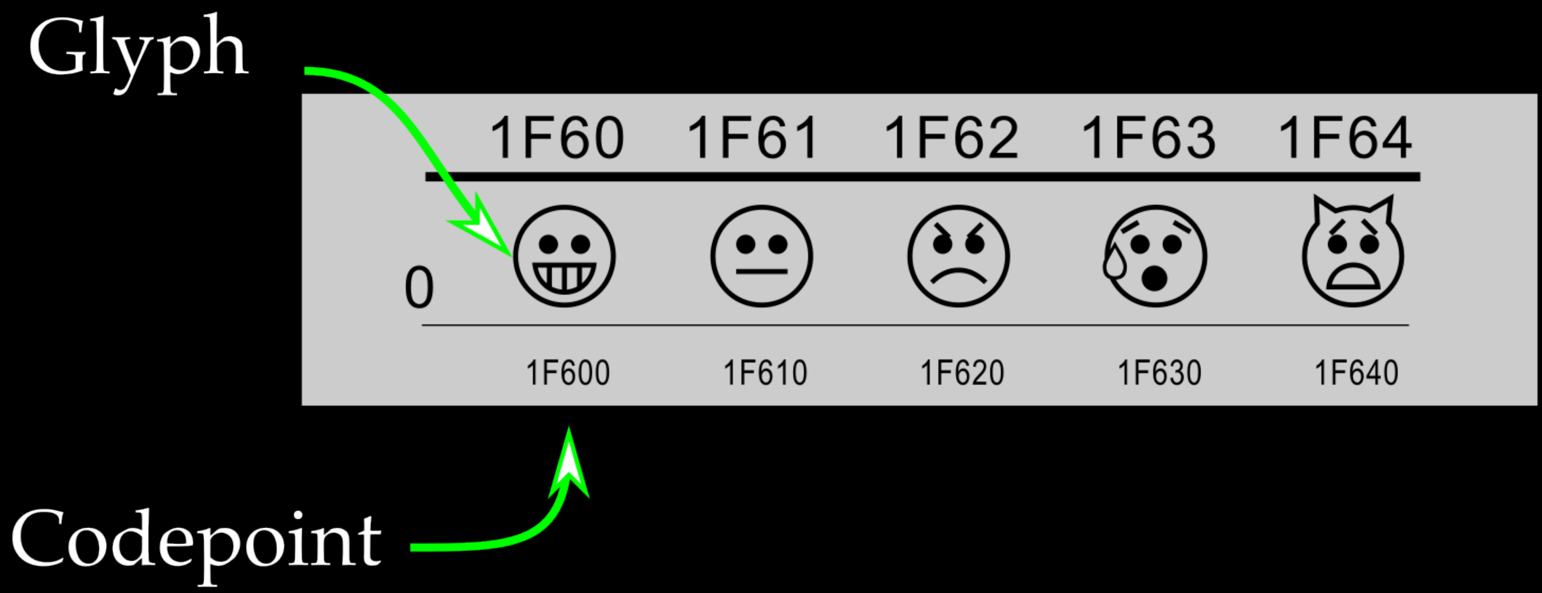
Example

- Character - SUPERSCRIPT TWO
- Glyph - ^{2}
- Code point - U+178
- Encoding - `b'\xc2\xb2'` (UTF-8) `b'\xb2'` (windows-1252)

Code Points

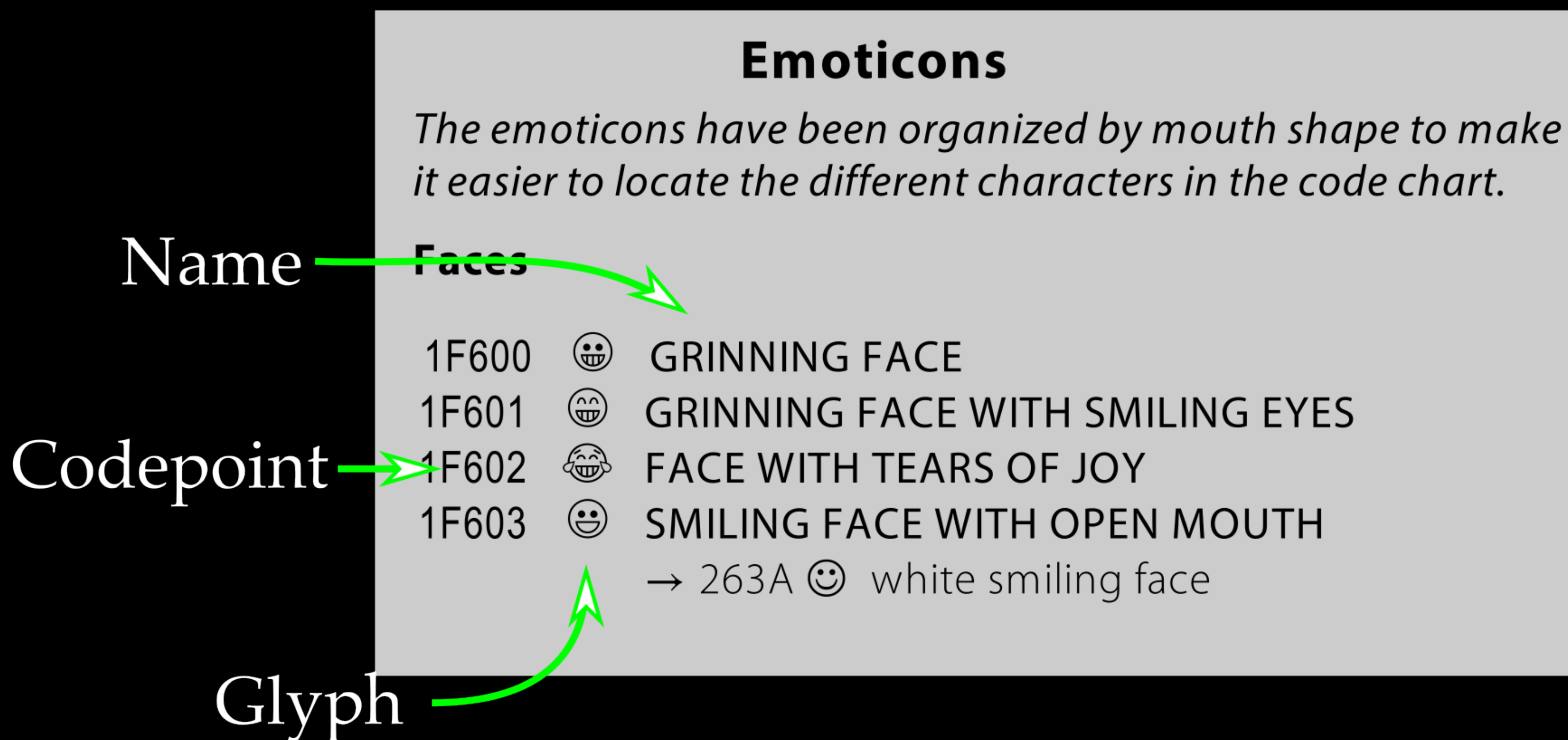
http://unicode.org has code charts that map letters to a Unicode *character code*.

Unicode Chart



From https://unicode.org

Unicode Chart



Unicode Chart

Name	`'\N{GRINNING FACE}'`
Codepoint	`'\U0001f600'`
Glyph	`'😀'`
UTF-8	`b'\xf0\x9f\x98\x80'.decode('utf8')`

Python 3

Everything is stored as Unicode in 2 (UCS-2 `sys.maxunicode == 65535`) or 4 bytes (UCS-4 `sys.maxunicode == 1114111`). (2-4x Python 2)

Bytes are NOT Python 2 strings. Rather arrays of integers.

Encodings

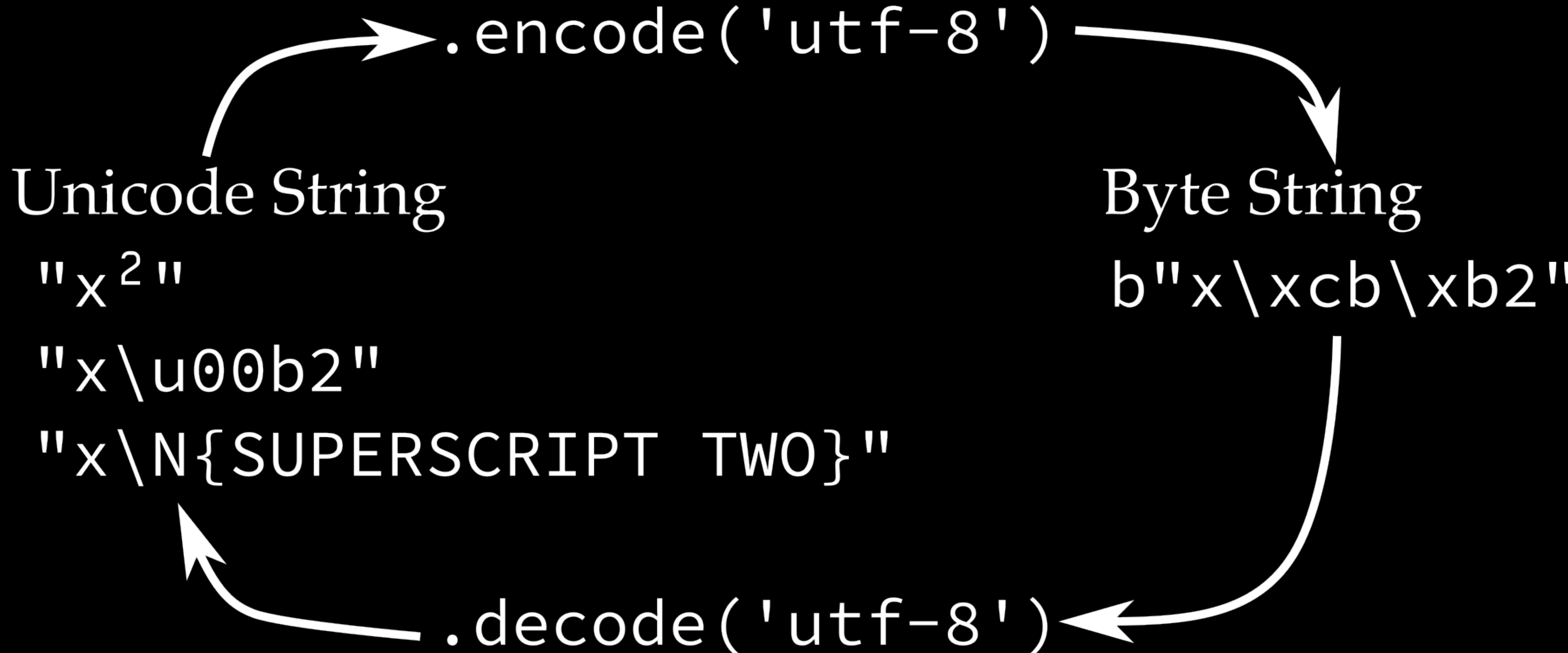
Encodings map bytes to code points. Encoding is not Unicode number!

Encode vs Decode

Unicode is *encoded* to bytes. Bytes are *decoded* to Unicode.

```
>>> c = '\u2126'    # Code point
>>> c
'Ω'
>>> c.encode('utf16')
b'\xff\xfe&!'
>>> c.encode('utf8')
b'\xe2\x84\xa6'
>>> b'\xe2\x84\xa6'.decode('utf8')
'Ω'
```

Unicode Encoding & Decoding



Errors

Encode error means the encoding doesn't support the character:

```
>>> c = '\u2126'    # Code point
>>> c
'Ω'
>>> c.encode('ascii')
Traceback (most recent call last):
  ...
UnicodeEncodeError: 'charmap' codec can't encode character
'\u2126' in position 0: character maps to <undefined>
```

Errors

ASCII and Windows-1252 fail, but CP949 (Korean) doesn't:

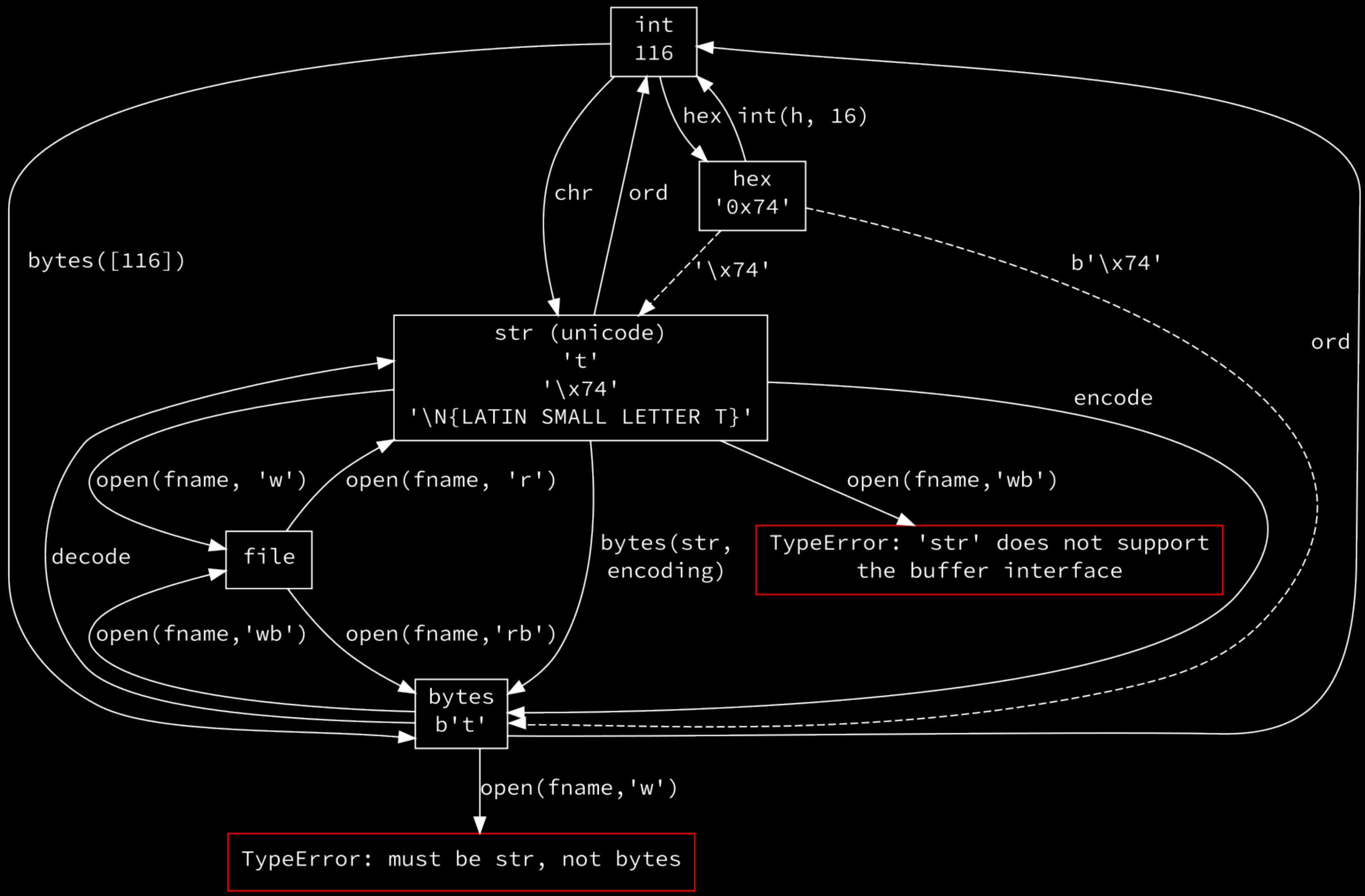
```
>>> c.encode('windows-1252')
Traceback (most recent call last):
  ...
UnicodeEncodeError: 'charmap' codec can't encode character
'\u2126' in position 0: character maps to <undefined>

>>> c.encode('cp949')  # Korean
b'\xa7\xd9'
```

Errors

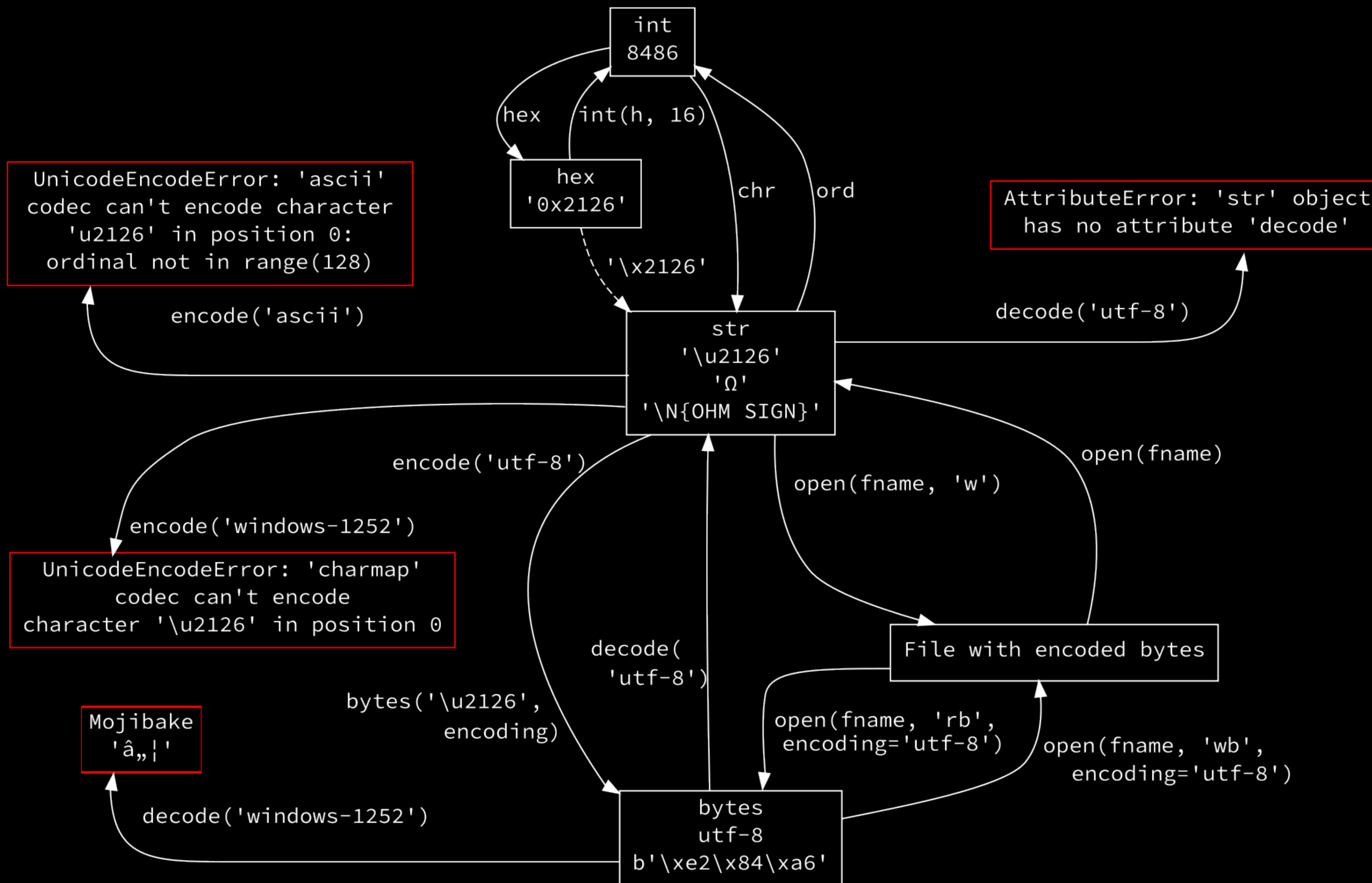
Decode error is going from bytes to encoding that doesn't support that byte sequence:

```
>>> kor = c.encode('cp949')  # Korean
>>> kor.decode('utf8')       # Bad!
Traceback (most recent call last):
  ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xa7
in position 0: invalid start byte

>>> kor.decode('cp1026')  # Turkish, also bad! (mojibake)
'xR'
```

Unicode Encoding & Decoding



Unicode Encoding & Decoding



Assignment

unicode_test.py

Unicode in Files

Text Files

Text files are read using system encoding. Best practice to be explicit.

```
>>> import locale
>>> locale.getpreferredencoding(False)
'UTF-8'
```

Explicit

Specify encoding:

```
>>> data = 'this is Ohm: Ω'
>>> with open('/tmp/ohm.kor', 'w',
...                encoding='cp949') as fout:
...         fout.write(data)
```

Explicit

When reading file, specify encoding (Korean is not the default):

```
>>> data2 = open('/tmp/ohm.kor', 'r', encoding='cp949').read()
>>> data2
'this is Ohm: Ω'

>>> data3 = open('/tmp/ohm.kor', 'r').read()
Traceback (most recent call last):
  ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xa7
in position 13: invalid start byte
```

Explicit

With binary, don't specify encoding (but provide bytes):

```
>>> data = 'this is Ohm: Ω'.encode('utf8')
>>> with open('/tmp/ohm2.utf8', 'wb',
...           encoding='utf8') as fout:
...     fout.write(data)
Traceback (most recent call last):
  ...
ValueError: binary mode doesn't take an encoding
argument
```

Assignment

`unifile_test.py`

Unicode Identifiers (PEP 3131 - 3.0)

Unicode Variables

By using identifiers in their native language, code clarity and maintainability of the code among speakers of that language improves

PEP 3131

Unicode Variables

```
>>> Ω_val = 10
>>> Ω_val
10
```

Unicode Variables

Still can't start with a number:

```
>>> 2Ω_val = 10
Traceback (most recent call last):
  ...
    2Ω_val = 10
         ^
SyntaxError: invalid syntax
```

String Formatting (PEP 3101 - 3.0)

String formatting

C-like:

```
>>> "%s %s" %('hello', 'world')
'hello world'
```

PEP 3101 adds `.format` method:

```
>>> "{0} {1}".format('hello', 'world')
'hello world'
```

Some Expressions Allowed

Attribute and index access only:

```
>>> '{[age]}'.format({'age': 50})
'50'

>>> class Person: pass
>>> p = Person()
>>> p.age = 50
>>> '{.age}'.format(p)
'50'
```

Some Expressions Allowed

Can't invoke methods:

```
>>> '{.upper()}'.format('matt')
Traceback (most recent call last):
  ...
AttributeError: 'str' object has no
attribute 'upper()'
```

Specify Position

```
>>> '{0} {0} {0} {1}'.format("na",
...                          "hey Jude")
'na na na hey Jude'
```

String Formatting

Language:

```
:[[fill]align][sign][#][0][width][,][.precision][type]
```

String Formatting

```
:[[fill]align][sign][#][0][width][,][.precision][type]
```

Fill character

Alignment

- `<` Left align
- `>` Right align
- `=` Pad after sign
- `^` Center align

Minimum Width

Thousands sep

Float digits (or max string length)

Sign

- `+` All nums
- `-` Neg nums
- Space Space pos, sign neg

`#` Prefix binary, octal, hex (0b,0o,0x)

`0` Zero padding (default space)

String or numeric

- `s` String (default)
- `r` Repr

- `b` Binary
- `c` Character
- `d` Decimal (default num)
- `o` Octal
- `x` Hex (lowercase)
- `X` Hex (uppercase)

- `e` Exponent (lowercase)
- `E` Exponent (uppercase)
- `f` Fixed point
- `g` General
- `n` Locale specific general
- `%` Percentage

Examples

Format a string in the center of 12 characters surrounded by *:

```
>>> "Name: {:*^12}".format("Ringo")
'Name: ***Ringo****'
```

Format a percentage using a width of 10, one decimal place and the sign before the width padding:

```
>>> "Percent: {:=10.1%}".format(-44./100)
'Percent: -    44.0%'
```

Binary and hex conversions:

```
>>> "Binary: {:b}".format(12)
'Binary: 1100'

>>> "Hex: {:x}".format(12)
'Hex: c'
```

See http://pyformat.info

Assignment

format_test.py

Literal String Interpolation (PEP 498 - 3.6)

Fstrings

The existing ways of formatting [strings] are either error prone, inflexible, or cumbersome.

PEP 498

Progression

```
>>> coin = 'bitcoin'
>>> price = 15690
>>> 'Coin: %s Price: %s' % (coin, price)
'Coin: bitcoin Price: 15690'

>>> 'Coin: {} Price: {}'.format(coin, price)
'Coin: bitcoin Price: 15690'

>>> f'Coin: {coin} Price: {price}'
'Coin: bitcoin Price: 15690'
```

Fstrings

If you precede a string literal with an `f` you can put an *expression* inside the `{}`

Fstrings

Things that you can return are *expressions*

```
>>> def to_spanish(word):
...     return 'uno' if word == 'one' else '?'
>>> val = 'one'
>>> f'Eng: {val} Es: {to_spanish(val)}'
'Eng: one Es: uno'
```

Fstrings

Can also format following :

```
>>> val = 12
>>> f"Binary: {val:b}"
'Binary: 1100'

>>> f"Hex: {val:x}"
'Hex: c'
```

Fstrings

Can use with raw strings, but not bytes or explicit Unicode literals (PEP 414)

Fstrings

Gotcha, careful with backslashes:

```
>>> f"Newline: {'\n'}"
Traceback (most recent call last):
  ...
SyntaxError: f-string expression part cannot include a
backslash

>>> nl = '\n'
>>> f"Newline: {nl}"
'Newline: \n'
```

Fstrings

Also faster!

```
>>> from timeit import timeit
>>> vars = "coin = 'bitcoin'; price=15690"
>>> timeit("'Coin: %s Price: %s' % (coin, price)", vars) # doctest:
+SKIP
0.3335153189837001
>>> timeit("'Coin: {} Price: {}'.format(coin, price)", vars) #
doctest: +SKIP
0.5271301100146957
>>> timeit("f'Coin: {coin} Price: {price}'", vars) # doctest: +SKIP
0.2377205429947935
```

Assignment

fstring_test.py

Explicit Unicode Literals

(PEP 414 - 3.3)

Explicit Unicode Literals

All strings are Unicode in Python 3 (implicitly).

Explicit Unicode Literals

the requirement to change the spelling of every Unicode literal in an application (regardless of how that is accomplished) is a key stumbling block for porting [Python 2 to 3] efforts.

PEP 414

Explicit Unicode Literals

If you are writing only Python 3, you can ignore this

Explicit Unicode Literals

```
>>> a = u"Unicode!"
>>> b = "Unicode!"
>>> a == b
True
```

Numbers

Integer Division (PEP 238 - 3.0)

Division

In Python 2, `x/y` would work for floats, but return the floor (int) for integers (floor division)

Division

In Python 3 `/` does *true division* (`__truediv__`) and `//` does *floor division* (`__floordiv__`)

Division

```
>>> 2 / 3
0.6666666666666666
>>> 2 // 3
0
>>> 2.0 / 3.0
0.6666666666666666
>>> 2.0 // 3.0
0.0
```

Dunder

```
>>> (2).__truediv__(3)
0.6666666666666666

>>> (2).__floordiv__(3)
0
```

Unify Longs & Ints (PEP 237 - 3.0)

Longs

There is also the general desire to hide unnecessary details from the Python user when they are irrelevant for most applications... It makes sense to extend this convenience to numbers.

PEP 237

Precision

Python supports arbitrary precision ints. Limited by memory.

Storage

```
>>> import sys
>>> size = 0
>>> for i in range(100):
...     s = sys.getsizeof(2**i)
...     if s > size:
...         print(f'Num:{2**i} (2**{i}) bytes:{s}')
...         size = s
Num:1 (2**0) bytes:28
Num:1073741824 (2**30) bytes:32
Num:1152921504606846976 (2**60) bytes:36
Num:1237940039285380274899124224 (2**90) bytes:40
```

Details

In `Include/longintrepr.h` and `Objects/longobject.c`

Rounding

Rounding

The `round()` function rounding strategy and return type have changed. Exact halfway cases are now rounded to the nearest even result instead of away from zero. (For example, round(2.5) now returns 2 rather than 3.)

What's New in Python 3.0

Rounding

```
>>> round(2.5)
2
>>> round(3.5)
4
```

Rounding

To nearest even number is called *banker's rounding*. It tries to eliminate bias to round high.

Rounding

Note The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68`. This is not a bug: it’s a result of the fact that most decimal fractions can’t be represented exactly as a float.

Python docs

Floats

Lack precision (but do the *right thing*):

```
>>> round(.05, 1), round(.15, 1)
(0.1, 0.1)
```

Underscores in numeric literals (PEP 515 - 3.6)

Underscores in Numbers

Readability counts.

`import this`

Underscores in Numbers

Intent is to group decimals by thousands or hex by words.

Underscores in Numbers

```
>>> 120_000_000 - 3_000_000
117000000
>>> 0xDEAD_BEEF
3735928559
```

Underscores in Numbers

Be careful

```
>>> 1_2_3_45_6
123456
```

Assignment

num_test.py

Statistics
(PEP 450 - 3.4)

Stats

Even simple statistical calculations contain traps for the unwary... This problem plagues users of many programming language, not just Python, as coders reinvent the same numerically inaccurate code over and over again

PEP 450

Floating Point Issues

Adding a constant should not change the variance:

```
>>> def var(nums):
...     n = len(nums)
...     total = sum(x**2 for x in nums) - (sum(nums)**2)/n
...     return total/(n-1)

>>> data = [1, 2, 3, 4, 5]
>>> var(data)
2.5
>>> var([x+1e13 for x in data])
-17179869184.0
```

Floating Point Issues

Adding a constant should not change the variance:

```
>>> from statistics import variance
>>> data = [1, 2, 3, 4, 5]
>>> variance(data)
2.5
>>> variance([x+1e13 for x in data])
2.5
```

Functions

`harmonic_mean`, `math`, `mean`, `median`, `median_grouped`, `median_high`, `median_low`, `mode`, `pstdev`, `pvariance`, `stdev`, `variance`

Assignment

stat_test.py

Classes

super (PEP 3135)

Super

Built-in mechanism to get access to parent methods. Python 3 syntax cleaned up.

Super

Options for Object Oriented Programming:

- Defer to parent class (do nothing)
- Override (or overload) method (implement method)
- Specialize (use `super`)

Super

Don't do this:

```
class Animal:
    def talk(self):
        return 'Sound'

class Cat(Animal):
    def talk(self):
        parent = Animal.talk(self)
        return '{} and Purr'.format(parent)
```

Super

Do this:

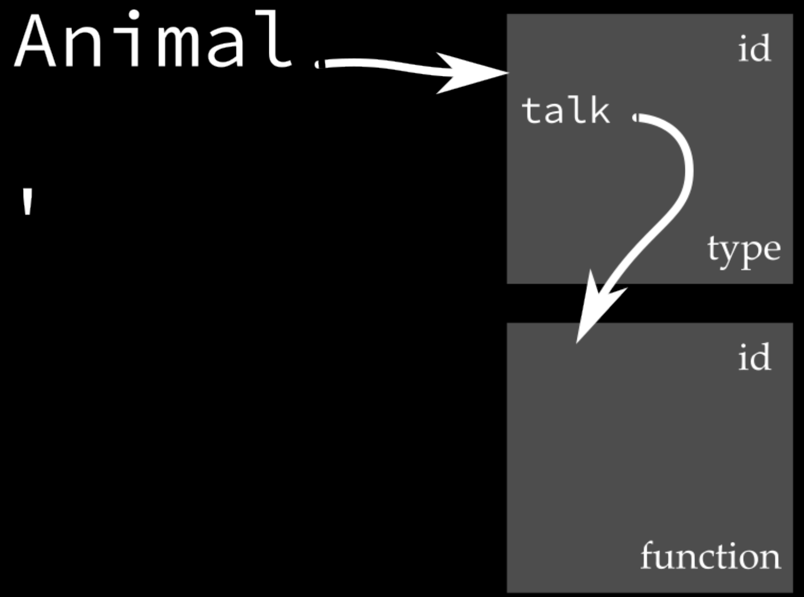
```python
class Animal:
    def talk(self):
        return 'Sound'

class Cat(Animal):
    def talk(self):
        parent = super().talk()
        return '{} and Purr'.format(parent)
```

Super

Code

```
class Animal:
    def talk(self):
        return 'Sound'
```

Objects

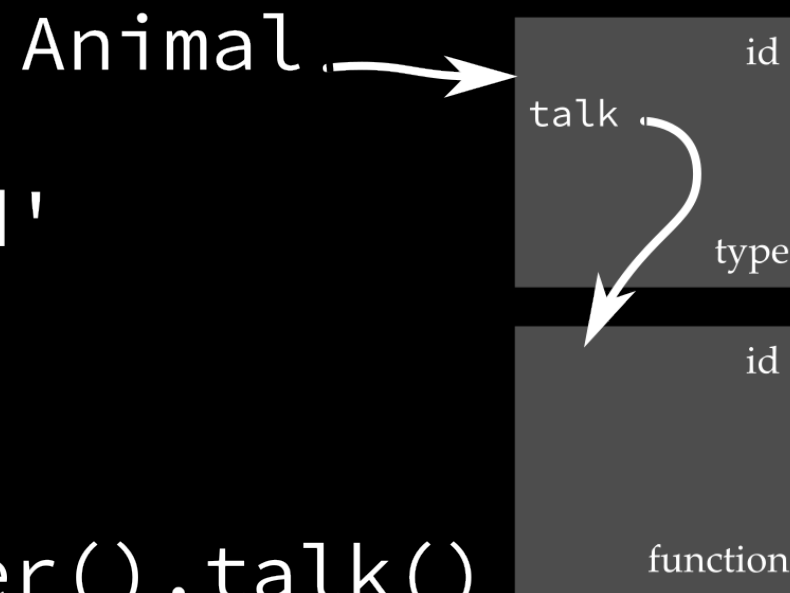


Super

Code

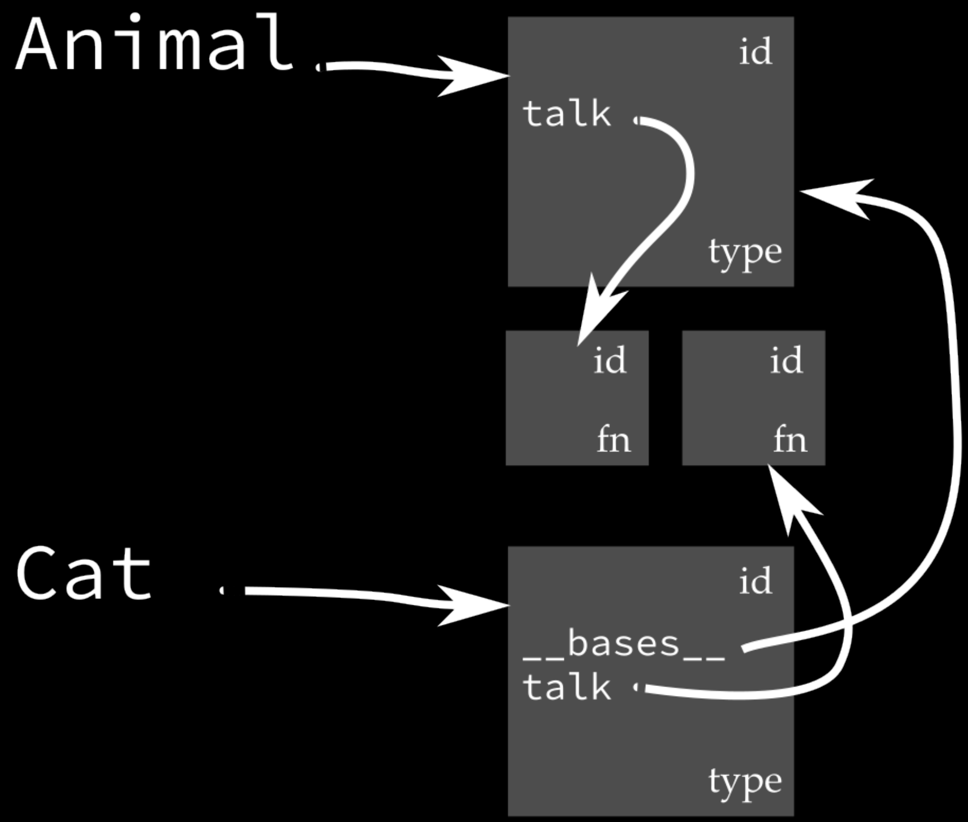
```
class Animal:
    def talk(self):
        return 'Sound'

class Cat(Animal):
    def talk(self):
        parent = super().talk()
        return parent
```

Objects

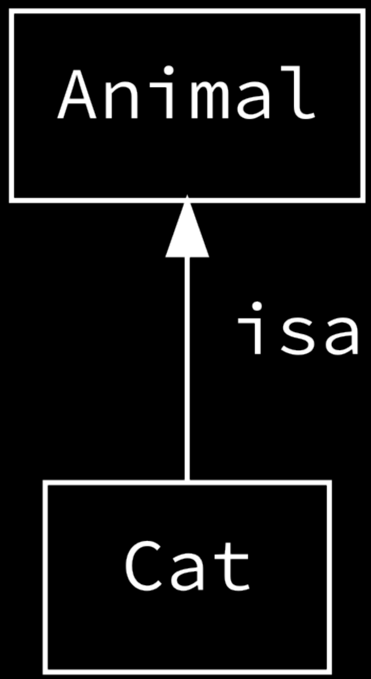


Super

Objects



Super



`super`

- No need to change call if change parent class
- Useful in multiple inheritance (follows `__mro__`)
- Need to be consistent with using only `super`

MRO

Method Resolution Order

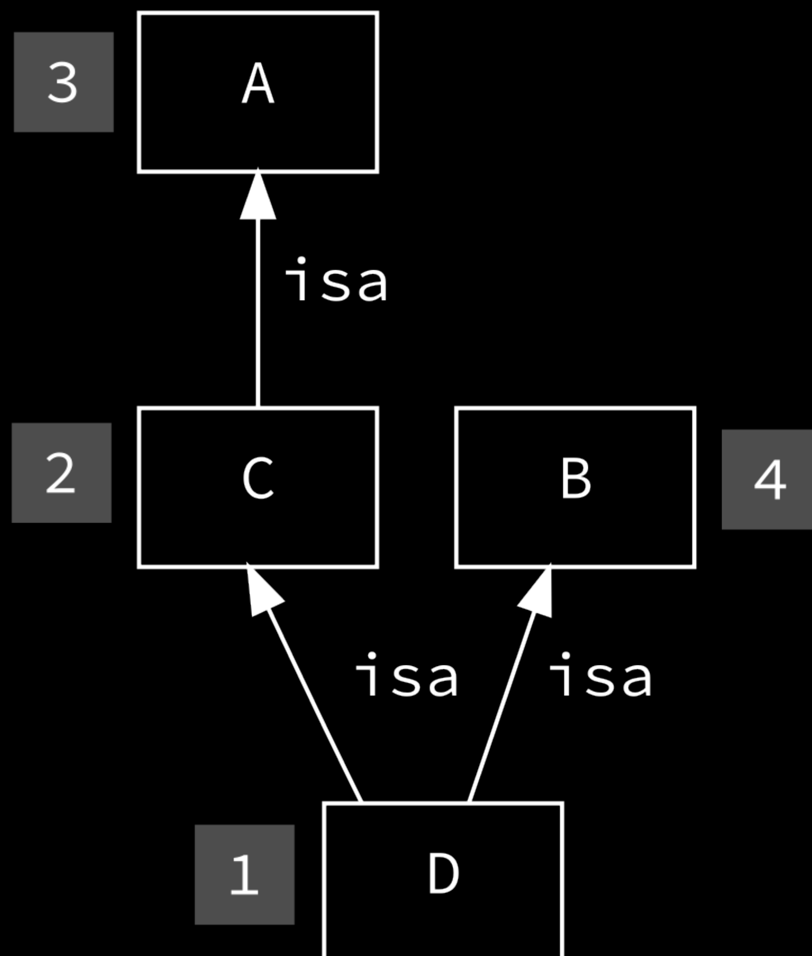
```
>>> class A:
...     def hi(self):
...         return "HI"
>>> class B:
...     def hi(self):
...         return "hello"
>>> class C(A): pass
>>> class D(C, B): pass
```

MRO

We can inspect this order by calling `mro` on the class:

```
>>> D.mro()
[<class '__main__.D'>, <class
'__main__.C'>, <class '__main__.A'>,
<class '__main__.B'>, <class 'object'>]
```

Super



MRO

With Python 3 follows C3 linearization algorithm. Depth first generally. If diamond pattern, use last occurrence of repeated parents.

http://www.webcom.com/haahr/dylan/linearization-oopsla96.html
http://python-history.blogspot.com/2010/06/method-resolution-order.html

MRO

Diamond pattern Method Resolution Order

```
>>> class A(dict):
...     def hi(self):
...         return "HI"
>>> class B(dict):
...     def hi(self):
...         return "hello"
>>> class C(A): pass
>>> class D(C, B): pass
```

MRO

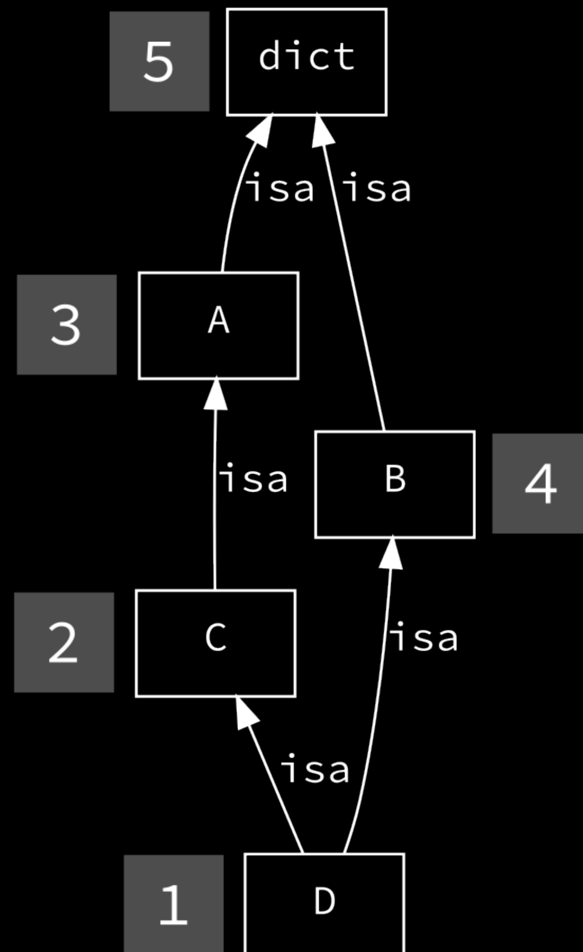
- L[D] = D + L[C] + L[B]
- L[D] = D + C + L[A] + B + L[dict]
- L[D] = D + C + A + L[dict] + B + L[dict]
- L[D] = D + C + A + B + dict

MRO

Diamond. Goes to `B` before `dict`:

```
>>> D.mro()
[<class '__main__.D'>, <class
'__main__.C'>, <class '__main__.A'>,
<class '__main__.B'>, <class 'dict'>,
<class 'object'>]
```

Super



Assignment

super_test.py

Matrix multiplication (PEP 465 - 3.5)

Matrix Multiplication

In numerical code, there are two important operations which compete for use of Python's `*` operator: elementwise multiplication, and matrix multiplication.

PEP 465

Matrix Multiplication

```
>>> import numpy as np
>>> a = np.array(range(10))
>>> b = np.array(range(10))
>>> sum(x*y for x, y in zip(a, b))
285
>>> a @ b
285

>>> a * 10
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Matrix Multiplication

```
>>> class Arr:
...     def __matmul__(self, other):
...         return 42

>>> a = Arr()
>>> b = Arr()
>>> a @ b
42
```

Assignment

mul_test.py

Key sharing dictionary (PEP 412 - 3.3)

Key Sharing

allows dictionaries which are used as attribute dictionaries (the __dict__ attribute of an object) to share keys with other attribute dictionaries of instances of the same class.

PEP 412

Result

These dictionaries are typically half the size of the current dictionary implementation.

Benchmarking shows that memory use is reduced by 10% to 20% for object-oriented programs with no significant change in memory use for other programs.

PEP 412

Functions

Keyword only arguments (PEP 3102 - 3.0)

Keyword only

One can easily envision a function which takes a variable number of arguments, but also takes one or more 'options' in the form of keyword arguments. Currently, the only way to do this is to define both a varargs argument, and a 'keywords' argument (`**kwargs`), and then manually extract the desired keywords from the dictionary.

PEP 3012

Keyword only

- Named args after `*vargs`
- Can use bare `*`

Keyword only

```
>>> def foo(*args, name='joe'):
...     return f'Hey {name}'

>>> foo()
'Hey joe'
>>> foo('matt')
'Hey joe'
>>> foo(name='matt')
'Hey matt'
```

Keyword only

```
>>> def foo2(*, name='joe'):
...     return f'Hey {name}'

>>> foo2()
'Hey joe'
>>> foo2('matt')
Traceback (most recent call last):
  ...
TypeError: foo2() takes 0 positional arguments but 1 was given
>>> foo2(name='matt')
'Hey matt'
```

Keyword only

```
>>> def foo3(*, name):
...     return f'Hey {name}'

>>> foo3()
Traceback (most recent call last):
  ...
TypeError: foo3() missing 1 required keyword-only argument: 'name'
>>> foo3('matt')
Traceback (most recent call last):
  ...
TypeError: foo3() takes 0 positional arguments but 1 was given
>>> foo3(name='matt')
'Hey matt'
```

Keyword only

Improves readability of functions by removing positional arguments.

```
send(404, 200, 100)
```

vs

```
send(code=404, amount=200, timeout=100)
```

Assignment

keyword_test.py

Annotations

(PEP 3107 - 3.0
PEP 484 - 3.5
PEP 526 - 3.6)

Lots of PEPS

- 3107 - Function annotations (vaguely hinted at typing, but generic)
- 482 - Overview of literature
- 483 - Theory
- 484 - Type hint standard
- 526 - Variable annotations
- 544 - Structural subtyping (3.7 draft)

Motivation

Introduces a provisional module to provide [syntax for function annotations] and tools, along with some conventions for situations where annotations are not available.

PEP 484

Motivation

This PEP aims at adding syntax to Python for annotating the types of variables (including class variables and instance variables), instead of expressing them through comments

PEP 526

Python Type Hints

You can document the types in your code. Has no effect at runtime! (not slower, not faster)

Static vs Dynamic

Types checked at compile time vs runtime

Benefits of Static Typing

- Aid comprehension for large code bases (better than comments)
- Catch bugs
- Autocompletion
- Refactoring

Annotations

aims to provide a single, standard way of specifying [a functions's parameters and return values]

PEP 3107

Annotations

Mark types for functions and classes. Does not actually check anything, need 3rd party tools (mypy) for that.

Syntax

Support for arguments (PEP 3107):

```python
def foo(a: expression, b: expression = 5):
    ...
```

Syntax

Support for * and ** arguments (PEP 3107):

```
def foo(*args: expression,
        **kwargs: expression):
    ...
```

Syntax

Support for return values (PEP 3107):

```
def sum() -> expression:
    ...
```

Syntax

No support for `lambda`

Syntax

From PEP 526

```
>>> name: str = 'Paul'
>>> name2: str      # No value
```

Example

```
>>> def sum2(x: 'num', y: int) -> float:
...     return x + y

>>> sum2.__annotations__
{'x': 'num', 'y': <class 'int'>, 'return':
<class 'float'>}
```

Variable Annotations

PEP 484 (used comments for variables). PEP 526 (3.6) updated:

```
>>> name = 'Matt'  # type: str
>>> other: str = 'Paul'
>>> middle: str    # No value

>>> __annotations__
{'name': <class 'str'>, 'other': <class 'str'>,
'middle': <class 'str'>}
>>> middle
Traceback (most recent call last):
  ...
NameError: name 'middle' is not defined
```

`typing` library

In Python 3.5. Adds support for `Any`, `Union`, `Tuple`, `Callable`, `TypeVar`, and `Generic`.

Use quoted strings for forward declarations.

Other Annotations

```
>>> from typing import Dict
>>> ages: Dict[str, int] = {'fred': 10}
```

Other Annotations

```
>>> from typing import List
>>> names: List[str] = ['fred', 'george']
```

Other Annotations

```
>>> from typing import Tuple
>>> person: Tuple[str, int, str] = \
...     ('fred', 10, 'USA')
```

Other Annotations

Callables should be `typing.Callable[params, ret]`:

```
>>> from typing import Callable
>>> def add(x, y):
...     return x + y

>>> def repeat(times: int,
...            fn: Callable[[int, int], int],
...            args: Tuple[int, int]) -> None:
...     for i in range(times):
...         fn(*args)
```

Other Annotations

PEP 526 has no support for `for` loops or `with` statement. PEP 484 support (from PEP):

```
with frobnicate() as foo:  # type: int
    # Here foo is an int
    ...

for x, y in points:  # type: float, float
    # Here x and y are floats
    ...
```

Other Annotations

Generators use `typing.Generator[yield_type, send_type, return_type]`. From PEP 484:

```
def echo_round() -> Generator[int, float, str]:
    res = yield
    while res:
        res = yield round(res)
    return 'OK'
```

Best Practices

- Use `typing.Callable` for function passing
- Use `typing.Any` to disregard type

Best Practices

Instead of returning `List[str]`, create a `AppendResult` variable. If result is tuple, consider making a named tuple.

Best Practices

For named tuples use `typing.NamedTuple`. Instead of:

```
>>> from collections import namedtuple
>>> Person = namedtuple('Person', 'name age country')
```

Do:

```
>>> from typing import NamedTuple
>>> class Person(NamedTuple):
...     name: str
...     age: int
...     country: str
```

Best Practices

Use `typing.Optional` when `None` may be returned
(`mypy --strict-optional` will warn you):

```
>>> from typing import Optional
>>> def find(name, peeps) -> Optional[Person]:
...     for person in peeps:
...         if person.name == name:
...             return person
...     return None
```

Best Practices

Use the `reveal_type` function of mypy (don't need to import it) when you aren't sure of a type:

```
# main.py
def add(x: int, y: float):
    res = x + y
    reveal_type(res)
    return res
```

Mypy example

Adding typing to

https://github.com/mattharrison/pycon-beg-markov-2017/blob/master/markov.py

```
(env) $ pip install mypy
(env) $ git clone
https://github.com/mattharrison/pycon-beg-markov-2017
(env) $ cd pycon-beg-markov-2017
(env) $ mypy markov.py
# no output!
```

Mypy example

Gradual typing means `mypy` ignores code without annotations (hence no output on previous slide)

Mypy example

Run with `--strict` to get more output:

```
(env) $ mypy --strict markov.py
markov.py:36: error: Function is missing a type annotation
markov.py:39: error: Call to untyped function "get_table" in typed context
markov.py:42: error: Function is missing a type annotation
markov.py:54: error: Function is missing a type annotation
markov.py:56: error: Need type annotation for variable
markov.py:71: error: Function is missing a type annotation
markov.py:79: error: Function is missing a type annotation
markov.py:92: error: Function is missing a type annotation
markov.py:105: error: Call to untyped function "Markov" in typed context
markov.py:106: error: Call to untyped function "repl" in typed context
markov.py:116: error: Call to untyped function "main" in typed context
```

Hints

- Start from outside (functions you first call)
- Run `mypy file.py`
- Rinse, repeat
- Run `mypy --strict` if you want bonus points

Diffs

```
import sys
+from typing import Dict, List


+TableResult = Dict[str, Dict[str, int]]
+
 class Markov:

-    def __init__(self, data, size=1):
-        self.tables = []
+    def __init__(self, data: str, size: int=1) -> None:
+        self.tables : List[TableResult] = []
         for i in range(size):
```

Diffs

Because I reused the `result` variable, mypy needed this comment:

```
-    def predict(self, data_in):
+    def predict(self, data_in: str) -> str:
         table = self.tables[len(data_in) - 1]
         options = table.get(data_in, {})
         if not options:
             raise KeyError()
         possible = ''
-        for result, count in options.items():
+        for result, count in options.items(): # type: str, int
             possible += result*count
         result = random.choice(possible)
         return result
```

Diffs

```
-def get_table(line, numchars=1):
-    results = {}
+def get_table(line: str, numchars: int=1) ->
TableResult:
+    results: TableResult = {}
     for i, char in enumerate(line):
```

Result

- Code is more clear
- Found possible bug (reusing `result` variable)

Assignment

annotate_test.py

3rd Party Annotation Tooling

3rd Party Tooling

- MonkeyType (Instagram) - type hints via tracing
- PyAnnotate (Dropbox) - type hints via tracing
- Pytype (Google) - static type checking

MonkeyType

- Run code (not static, might need driver file)
- Collect type info in Sqlite file
- Annotate code with type info

Code Runner

```
# runtests.py
import doctest
import markov
doctest.testmod(markov)
```

MonkeyType

```
$ pip install monkeytype
$ monkeytype run runtests.py
$ monkeytype stub markov
```

MonkeyType

```
# stub output
from typing import Dict

def get_table(line: str, numchars: int = 1) -> \
             Dict[str, Dict[str, int]]: ...

def test_predict(m: Markov, num_chars: int,
                 start: str, size: int = 1) -> str: ...

class Markov:
    def __init__(self, data: str, size: int = 1) -> None: ...
    def predict(self, data_in: str) -> str: ...
```

PyAnnotate

- Run code (not static, might need driver file)
- Collect type info in JSON file
- Annotate code with type info

PyAnnotate

```
$ pip install pyannotate
$ python driver.py
$ pyannotate -w markov.py
```

PyAnnotate

```
# driver.py
import doctest
import markov
from pyannotate_runtime import collect_types

if __name__ == '__main__':
    collect_types.init_types_collection()
    with collect_types.collect():
        doctest.testmod(markov)
    collect_types.dump_stats('type_info.json')
```

PyAnnotate output

```
Refactored markov.py
--- markov.py      (original)
+++ markov.py      (refactored)
@@ -29,17 +29,21 @@
 import sys
+from typing import Optional
+from typing import Dict


 class Markov:
     def __init__(self, data, size=1):
+        # type: (str, int) -> None
         self.tables = []
         ...
```

PyAnnotate output

```
def predict(self, data_in):
+         # type: (str) -> Optional[str]
          table = self.tables[len(data_in) - 1]
          ...


 def get_table(line, numchars=1):
+     # type: (str, int) -> Dict[str, Dict[str, int]]
      results = {}
      ...

 def test_predict(m, num_chars, start, size=1):
+     # type: (Markov, int, str, int) -> str
      res = [start]
      ...
```

PyAnnotate

Currently (Jan 2018) supports 2.7 style annotations.

mypy

Supports:

- 3x style annotations
- Comment (2.7) annotations
- Stub files (`.pyi`) for code you can't change

Stub files

Use `stubgen.py` to create stub files. Value of just having stub files with `Any` is that you can validate that the functions and methods you call exist.

mypy

```
$ pip install mypy
$ python -m mypy markov.py  #  mypy markov.py
markov.py:38: error: Need type annotation for variable
markov.py:57: error: Need type annotation for variable
```

mypy

Lines 38 & 57:

```
self.tables = []
results = {}
```

MonkeyType didn't add annotations here :(

mypy

To remove error, change line 38 to:

```
self.tables: List[Dict[str, Dict[str, int]]] = []
```

mypy

Integrate with CI (using `--cobertura-xml-report`)

mypy

Use the `reveal_type` function (don't need to import it) when you aren't sure of a type:

```python
# main.py
def add(x: int, y: float):
    res = x + y
    reveal_type(res)
    return res
```

mypy

Use the `reveal_type` function when you aren't sure of a type:

```
$ mypy main.py
main.py:3: error: Revealed type is 'builtins.float'
```

Pytype

Need to install with Python 2. (Can't use `pip` (Jan 2018)).

Pytype

```
(env2)$ pytype -V 3.6 markov.py
File "markov.py", line 39: Stray type comment: (str, int) -> None
[ignored-type-comment]
File "markov.py", line 46: Stray type comment: (str) -> Optional[str]
[ignored-type-comment]
File "markov.py", line 59: Stray type comment: (str, int) -> Dict[str,
Dict[str, int]] [ignored-type-comment]
File "markov.py", line 77: Stray type comment: (Markov, int, str, int) ->
str [ignored-type-comment]
File "markov.py", line 91, in repl: Function __builtin__.input expects 0
arg(s), got 1 [wrong-arg-count]
  Expected: ()
  Actually passed: (_)
```

Pytype

Generate stub (on original file):

```
(env2)$ pytype -V 3.6 --output - markov.py
from typing import Any, Dict, Optional

argparse = ...  # type: module
random = ...  # type: module
sys = ...  # type: module

class Markov:
    def __init__(self, data: str, size: int = ...) -> None: ...
    def predict(self, data_in: str) -> Optional[str]: ...


def get_table(line: str, numchars: int = ...) -> Dict[str, Dict[str, int]]: ...
def main(args) -> None: ...
def repl(m, size = ...) -> Any: ...
def test_predict(m: Markov, num_chars: int, start: str, size: int = ...) -> str: ...
# Plus some errors...
```

Summary

Works in progress. Pytype throws errors. MonkeyType may throw errors.

Assignment

`annotate3rd_test.py`

Standard Library

Print Function

`print` function

In Python 3 `print` is no longer a statement, but a function

print

Python 3. Defaults to inserting new line and spaces

```
>>> print(1, "one")
1 one

>>> print(1, "one", sep="-", end="END!")
1-oneEND!>>>
```

Print Function

```
>>> print(1, "one", sep="-", end="END!")

1-oneEND!
```

Assignment

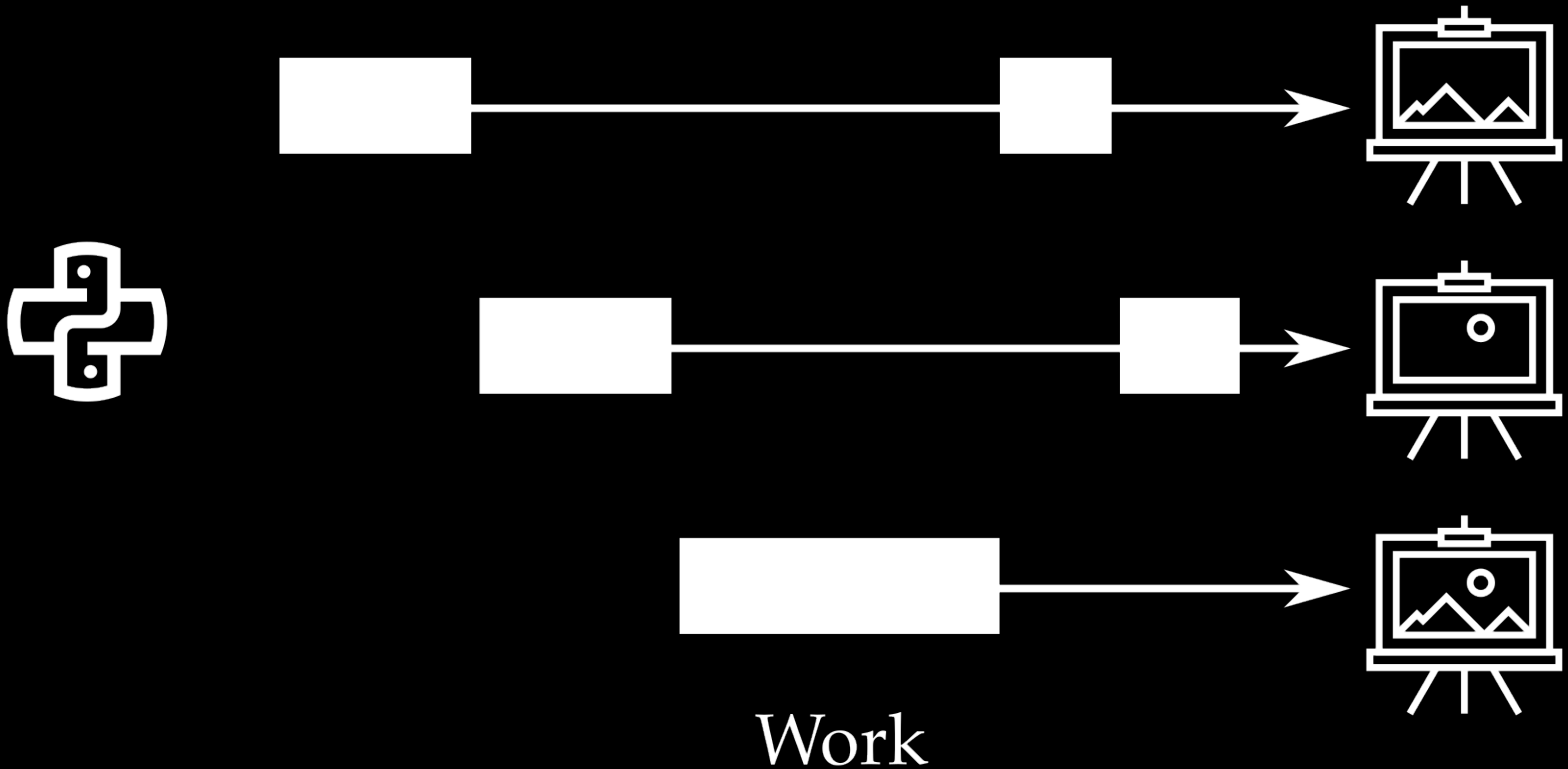
print_test.py

Asyncio
(PEP 525 - 3.6)

Terms

- **Concurrency:** Sharing resources (OS on CPU, one juggler multiple balls)
- **Parallelism:** Doing multiple things at once (execution) (Not possible on single CPU, multiple jugglers)
- **(Native) Thread:** OS construct for doing something (across CPU's)
- **Green Thread:** VM-level threads (lightweight but don't scale across CPU's)
- **Synchronous:** Wait til execution is done
- **Asynchronous:** Kick off execution and move on to something else

Concurrency



GIL

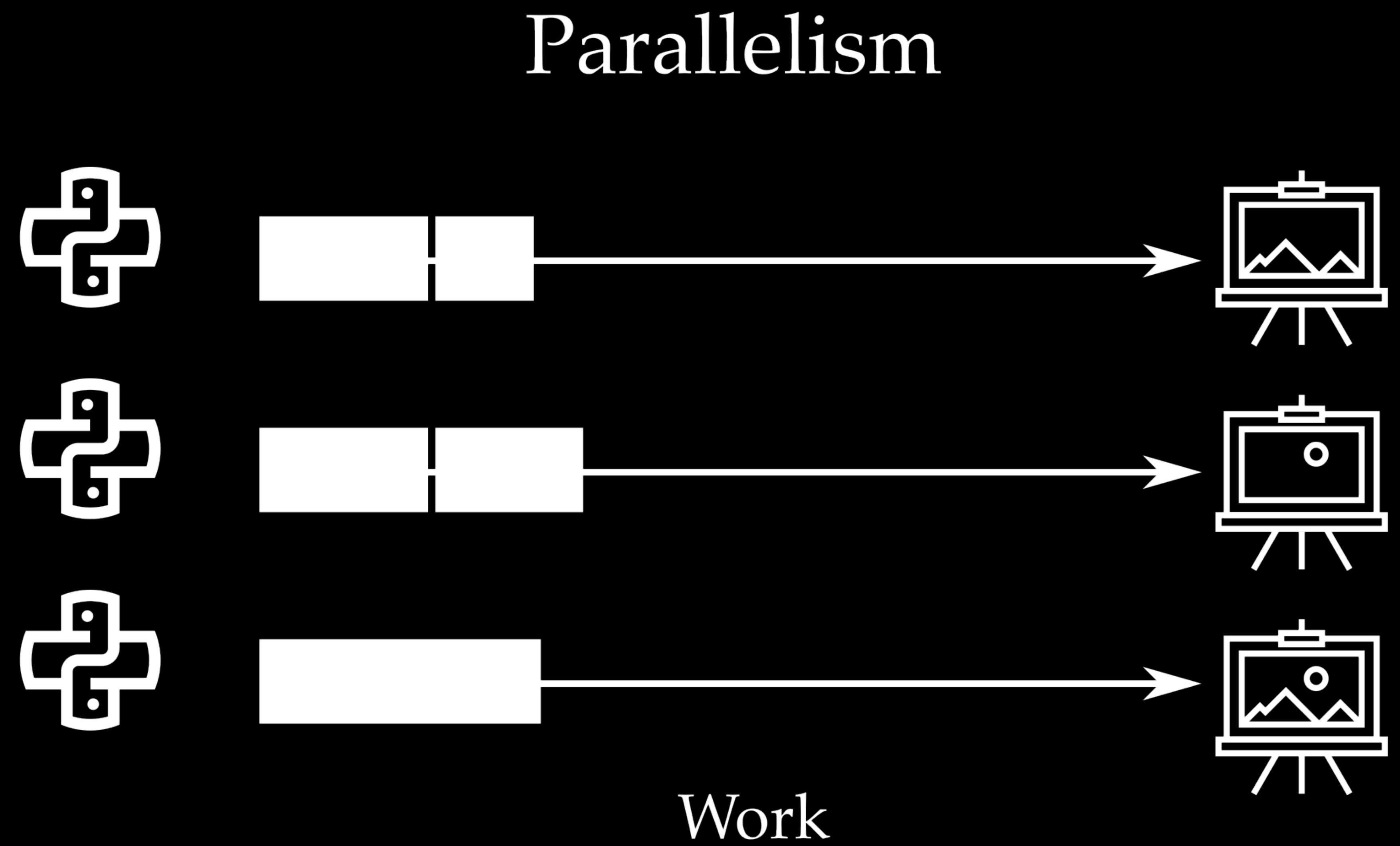
Global Interpreter Lock.

Plusses:

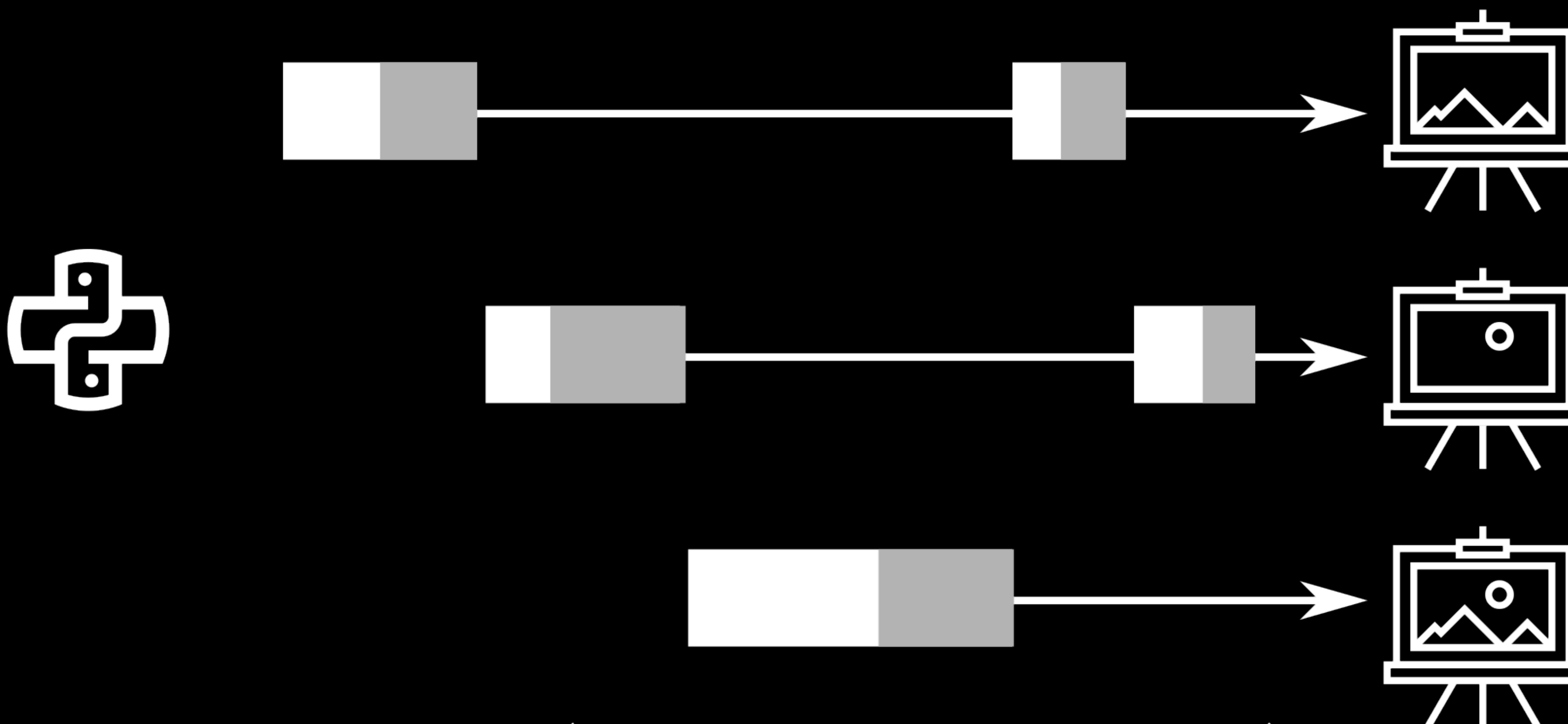
- Simplifies GC (not thread-safe)
- Avoid non-thread safe code with other threads.

Minuses:

- Only one native thread executes at a time
- CPU bound code slow

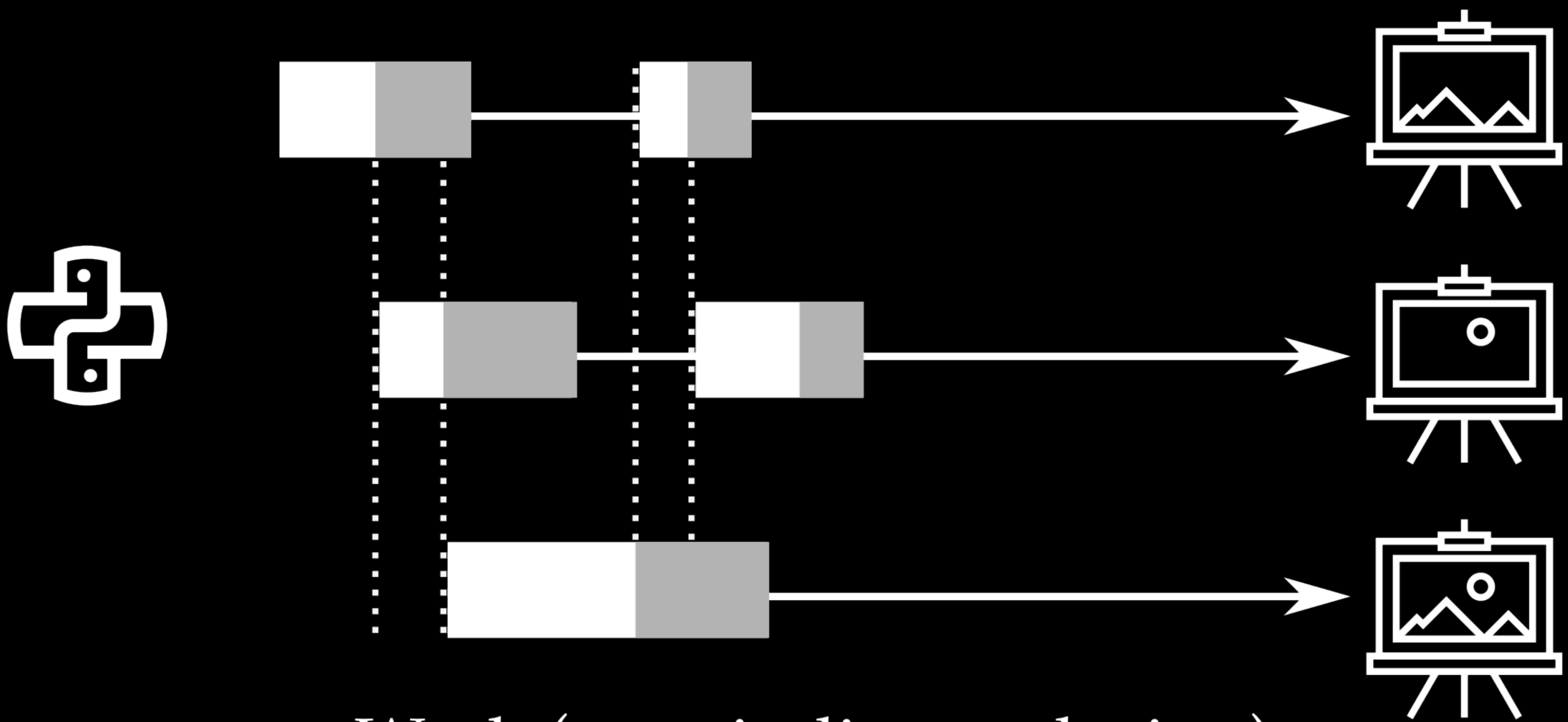

Parallelism
Work

Synchronous



Work (grey indicates drying)

Asynchronous



Work (grey indicates drying)

Painting Code

```
>>> class Canvas:
...     def paint(self, difficulty=100, dry=2):
...         # CPU heavy
...         for i in range(1,difficulty):
...             y = difficulty % i == 0
...         self.start = time.time()
...         self.end = self.start + dry
...
...     def is_dry(self):
...         if time.time() >= self.end:
...             print(f"{self} done")
...             return True
...         return False
```

Timing Decorator

```
>>> import functools

>>> def timing(fn):
...     @functools.wraps(fn)
...     def inner(*args, **kwargs):
...         start = time.time()
...         res = fn(*args, **kwargs)
...         print(f'{fn.__name__} took {time.time() -
start:.02f} secs')
...         return res
...     return inner
```

Synchronous

On my machine `run_paint took 6.03 secs`:

```
>>> import time
>>> def paint():
...     c = Canvas()
...     c.paint()
...     while not c.is_dry():
...         time.sleep(.5)

>>> @timing
... def run_paint():
...     paint()
...     paint()
...     paint()
```

Asynchronous

On my machine `run_async_paint` took 2.01 secs:

```
>>> import asyncio
>>> async def async_paint():
...     c = Canvas()
...     c.paint()
...     while not c.is_dry():
...         await asyncio.sleep(.5)

>>> @timing
... def run_async_paint():
...     loop = asyncio.get_event_loop()
...     fut = asyncio.gather(async_paint(),
...                          async_paint(),
...                          async_paint())
...     loop.run_until_complete(fut)
```

Basics

- Event loop - manages work
- Coroutines - suspendable functions
- Futures - result (may or may not be executed)
- Tasks - subclass of future that wraps a coroutine
- Context switch - change from one function to the next
- Blocking - wait until work is done before proceeding
- Non-blocking - can hand off control while running

Syntax

Python provides `async` and `await` for asynchronous programming, and the `asyncio` library as an implementation

Asyncio

Write asynchronous code in a sequential style

Why?

If you have a lot of IO (high latency not CPU), this scales better than threads or processes. (But whole stack needs to be Asyncio aware)

Components

- Functions need to suspend and resume
- Event loop keeps track of various functions and their states
- Long running CPU heavy tasks should routinely release the CPU so other functions may run
- Use async specific code or use executor

Developing Cooperative Multitasking

```
>>> def map(fn, seq):
...     res = []
...     for item in seq:
...         res.append(fn(item))
...     return res
```

(Hat tip Robert Smallshire)

Developing Cooperative Multitasking

Any async code must have at least one `yield` in it

```
>>> def async_map(fn, seq):
...     res = []
...     for item in seq:
...         res.append(fn(item))
...         yield
...     return res
```

Developing Cooperative Multitasking

Function is now a generator. We can advance it (`next`), and do other work in between

```
>>> gen = async_map(lambda x:x+2, range(3))
>>> next(gen)
>>> next(gen)
>>> 5 + 7  # other work
12
>>> next(gen)
>>> next(gen)
Traceback (most recent call last):
    ...
StopIteration: [2, 3, 4]
```

Developing Cooperative Multitasking

Can pull out results from exception:

```
>>> def runner(gen):
...     while 1:
...         try:
...             next(gen)
...         except StopIteration as e:
...             return e.value

>>> res = runner(async_map(lambda x:x+2, range(3)))
>>> res
[2, 3, 4]
```

Developing Cooperative Multitasking

Tasks and scheduler:

```
>>> class Task:
...     id = 1
...     def __init__(self, gen):
...         self.id = Task.id
...         Task.id += 1
...         self.gen = gen
```

Developing Cooperative Multitasking

Tasks and scheduler:

```
>>> from collections import deque
>>> class Scheduler:
...     def __init__(self):
...         self.tasks = deque()
...         self.results = {}
...         self.exceptions = {}
...     def add(self, gen):
...         self.tasks.append(Task(gen))
```

Developing Cooperative Multitasking

Tasks and scheduler:

```
...     def run(self):
...         while 1:
...             if not self.tasks: break
...             t = self.tasks.popleft()
...             try:
...                 print(f"run: {t.id}")
...                 next(t.gen)
...             except StopIteration as e:
...                 print(f"res: {t.id}: {e.value}")
...                 self.results[t.id] = e.value
...             except Exception as e:
...                 self.exception[t.id] = e
...             else:
...                 self.tasks.append(t)
```

Developing Cooperative Multitasking

Tasks and scheduler:

```
>>> g1 = async_map(lambda x:x+2, range(3))
>>> g2 = async_map(lambda x:x*3, range(4))
>>> s = Scheduler()
>>> s.add(g1); s.add(g2)
>>> s.run()
run: 1
run: 2
run: 1
run: 2
run: 1
run: 2
run: 1
res: 1: [2, 3, 4]
run: 2
run: 2
res: 2: [0, 3, 6, 9]
>>> s.results
{1: [2, 3, 4], 2: [0, 3, 6, 9]}
```

Asyncio

Code needs to be "infected". Everything a coroutine calls needs to be async. Need to iterate over results (use `await`)

Asyncio

To go to Python 3.6 `asyncio`:

- Replace `def async_` with `async def`
- Instead of `yield from` use `await`
- Use future to pass results

Developing Cooperative Multitasking

Tasks and scheduler:

```
>>> import asyncio
>>> async def amap(fut, fn, seq):
...     res = []
...     for item in seq:
...         res.append(fn(item))
...         await asyncio.sleep(0)
...     fut.set_result(res)
>>> s = asyncio.get_event_loop()
>>> f1 = s.create_future()
>>> t1 = amap(f1, lambda x:x+2, range(3))
>>> f2 = s.create_future()
>>> t2 = amap(f2, lambda x:x*3, range(4))
>>> f3 = asyncio.gather(t1, t2)
>>> s.run_until_complete(f3)
>>> f1.result()
[2, 3, 4]
>>> f2.result()
[0, 3, 6, 9]
>>> s.close()
```

Asyncio

- Code
- Event loop
  - create loop
  - call `.run_until_complete`
  - call `.close`

Coroutine

- declare coroutine with `async`
- `await` non-blocking code (`await asyncio.sleep(0)` facilitates context switch) (syntax can only be used in coroutine)
- To return value:
  - Pass in `asyncio.Future()`, use `fut.set_result(res)`
  - Use `return` to return results to another coroutine (or to `loop.run_util_complete(task).result()`)
- Needs to be *scheduled* or put in event loop (use `asyncio.wait` or `asyncio.gather` for multiple coroutines)

Future

Rather than calling `asyncio.Future`, call `loop.create_future` (event loops may provide an alternate implementation)

- `await f` - Wait until result arrives
- `f.set_result(val)` - Set result
- `f.set_exception(e)` - Set exception
- `f.add_done_callback(fn)` - Set a callback `fn(f)` to be called with done
- `f.exception()` - Return exception
- `f.result()` - Return result. Raises `InvalidStatError` if not done, `CancelledError` if cancelled. Use `res = yield from f` instead
- `f.cancel()` - Cancel future
- `f.done()`, `f.cancelled()` - Get status

Task

A task is responsible for executing a coroutine object in an event loop... Don’t directly create `Task` instances: use the `ensure_future()` function

docs.python.org

Task

`ensure_future` is more general (accepts an *awaitable* object) and idempotent, but does return a `Task`

Tips

- Use `loop.create_future` instead of `Future` to create a future
- Use `asyncio.gather` or `asyncio.ensure_future` to create `Tasks` from coroutines
- Use uvloop (3rd party) for faster loop implementation

Timeout

Can timeout (by seconds) a list of coroutines by using
`asyncio.wait(cos, timeout=1)`

Libraries

https://github.com/aio-libs

Debugging

Can use `pdb` (unlike threading, only one thread). Use `aioconsole` or `aiomonitor` as REPL

Testing

asynctest (on top of unittest), pytest-asyncio (pytest)

Assignment

async_test.py

Async Context Managers (PEP 492 - 3.5)

Protocol

- `__await__` - Coroutines are *awaitable*. You call `__await__` and iterate over results
- `__aiter__`, `__anext__` - Asynchronous iterators
- `__aenter__`, `__aexit__` - Asynchronous context managers.

Traditional Context Managers

If a class defines `__enter__` and `__exit__`, you can use it in a `with` statement

Traditional Context Managers

```
>>> class runner:
...     def __init__(self, item):
...         self.item = item
...     def __enter__(self):
...         self.item['running'] = True
...     def __exit__(self, ex, val, tb):
...         self.item['running'] = False

>>> item = {}
>>> with runner(item):
...     print(item['running'])
True
>>> print(item['running'])
False
```

Context Managers

Example of running external process async

https://github.com/arianon/panel/blob/master/panel/utils.py

Context Managers

```
# part of _AIOPopen

def __await__(self):
    if not self._proc:
        self._proc = yield from self._coro
    return self

async def __aenter__(self):
    return await self
```

Context Managers

Because `__aenter__` and `__aexit__` are coroutines, you can await inside if you need to.

Context Managers

Async timeout context manager

https://github.com/aio-libs/async-timeout

```
import asyncio

class Timeout:
    def __init__(self, timeout, loop):
        self.timeout = timeout
        self.loop = loop
        self.cancelled = False
        self.handler = None
```

Context Managers (2)

```
async def __aenter__(self):
    when = self.loop.time() + self.timeout
    self.task = get_task(loop)
    self.handler = self.loop.call_at(when, self.cancel)

async def __aexit__(self, exc, val, tb):
    if self.cancelled:
        raise asyncio.TimeoutError
    if self.handler:
        self.handler.cancel()
        self.handler = None
    self.task = None
```

Context Managers (3)

```
# from Timeout
    def cancel(self):
        self.task.cancel()
        self.cancelled = True

def get_task(loop):
    task = asyncio.Task.current_task(loop=loop)
    if task is None:
        if hasattr(loop, 'current_task'):
            task = loop.current_task()
    return task
```

Context Managers (4)

```python
async def run_timeout(loop):
    try:
        async with Timeout(2, loop):
            await asyncio.sleep(1)
            print("DONE")
    except asyncio.TimeoutError:
        print("TIMEOUT!")
    print("AFTER")

loop = asyncio.get_event_loop()
loop.run_until_complete(run_timeout(loop))
```

Assignment

`async_ctx_test.py`

Async Iterators
(PEP 492 - 3.5)

Protocol

- `__await__` - Coroutines are *awaitable*. You call `__await__` and iterate over results
- `__aiter__`, `__anext__` - Asynchronous iterators
- `__aenter__`, `__aexit__` - Asynchronous context managers.

Async Interator

```
class Arange:
    def __init__(self, start, end=None):
        if end is None:
            self.start = 0
            self.end = start
        else:
            self.start = start
            self.end = end
```

Async Interator (2)

```
def __aiter__(self):
    return self

async def __anext__(self):
    val = self.start
    if val >= self.end:
        raise StopAsyncIteration
    self.start += 1
    return val
```

Async Interator (3)

```
async def run_arange(loop):
    async for x in Arange(5):
        print(x)

loop = asyncio.get_event_loop()
loop.run_until_complete(run_arange(loop))
```

Assignment

`async_iter_test.py`

Async Generators
(PEP 525 - 3.6)

Async Generators

However, currently there is no equivalent concept for the asynchronous iteration protocol (`async for`). This makes writing asynchronous data producers unnecessarily complex, as one must define a class that implements `__aiter__` and `__anext__` to be able to use it in an `async for` statement.

Performance is an additional point for this proposal: in our testing of the reference implementation, asynchronous generators are 2x faster than an equivalent implemented as an asynchronous iterator.

PEP 525

Async Iterator + Generator

```python
class GenRange:
    def __init__(self, start, end=None):
        if end is None:
            self.start = 0
            self.end = start
        else:
            self.start = start
            self.end = end

    async def __aiter__(self):
        for i in range(self.start, self.end):
            yield i
```

Async Iterator + Generator (2)

```
async def run_genrange(loop):
    async for x in GenRange(5):
        print(x)

loop = asyncio.get_event_loop()
loop.run_until_complete(run_genrange(loop))
```

Async Generator

```python
async def gen_range(start, end=None):
    if end is None:
        end = start
        start = 0
    for i in range(start, end):
        yield i
```

Async Generator (2)

```
async def run_gen_range(loop):
    async for x in gen_range(5):
        print(x)


loop = asyncio.get_event_loop()
loop.run_until_complete(run_gen_range(loop))
```

Assignment

`async_gen_test.py`

Pathlib
(PEP 428 - 3.4
PEP 519 - 3.6)

Pathlib

- 428 - Adds pathlib
- 519 - Adds protocol for paths (stdlib support for pathlib)

Pathlib

```
>>> from pathlib import Path
>>> env = Path('/tmp/env')
>>> list(env.iterdir())
[PosixPath('/tmp/env/.Python'),
PosixPath('/tmp/env/bin'),
PosixPath('/tmp/env/include'),
PosixPath('/tmp/env/lib')]
```

Pathlib

Path manipulation

```
>>> m = env / 'missing'
>>> m
PosixPath('/tmp/env/missing')
>>> m.exists()
False
>>> with m.open('w') as fout:
...     fout.write('hi')
>>> m.parts
('/', 'tmp', 'env', 'missing')
>>> with m.open() as fin:
...     print(fin.read())
hi
>>> m.unlink()
```

Pathlib

Path manipulation

```
>>> py = m.parent/'bin'/'activate_this.py'
>>> py.root
'/'
>>> py.drive
''
>>> py.anchor
'/'
```

Pathlib

Path manipulation

```
>>> py.parent
PosixPath('/tmp/env/bin')
>>> py.name
'activate_this.py'
>>> py.suffix
'.py'
>>> py.stem
'activate_this'
>>> py.is_absolute()
True
>>> py.match('*.py')
True
```

Pathlib

Pure paths and *concrete paths*. We mostly deal with concrete paths (that have access to the file system). But you can make Windows or Posix pure paths on either system for manipulation.

Pathlib

Concrete paths allow system calls

```
>>> py.cwd()  # or Path.cwd()
PosixPath('/tmp')
>>> py.home() # or Path.home()
PosixPath('/Users/matt')
>>> py.stat()
os.stat_result(st_mode=33188, st_ino=43448159, st_dev=16777220, st_nlink=1,
st_uid=501, st_gid=0, st_size=1137, st_atime=1513792060, st_mtime=1513792060,
st_ctime=1513792060)
>>> Path('~').expanduser()
PosixPath('/Users/matt')
>>> sorted(Path.home().glob('*.py'))
[PosixPath('/Users/matt/__init__.py')]
>>> #sorted(Path.home().glob('**/*.py'))  # all py files
```

Assignment

path_test.py

enum
(PEP 435 - 3.4)

Enum

An enumeration is a set of symbolic names bound to unique, constant values. Within an enumeration, the values can be compared by identity, and the enumeration itself can be iterated over.

PEP 435

Enum

```
>>> from enum import Enum
>>> class Bike(Enum):
...     road = 1
...     mtn = 2
...     cross = 3
...     trike = 4
```

Enum

```
>>> for bike in Bike:
...     print(bike)
Bike.road
Bike.mtn
Bike.cross
Bike.trike

>>> bike == Bike.trike
True
```

Enum

Access by attribute, number, or name

```
>>> Bike.mtn
<Bike.mtn: 2>
>>> Bike(2)
<Bike.mtn: 2>
>>> Bike['mtn']
<Bike.mtn: 2>
```

Enum

Identity comparisons work

```
>>> Bike.mtn is Bike.mtn
True
```

Enum

Alternate construction

```
>>> Bike2 = Enum('Bike', 'road mtn cross trike')
>>> Bike2(2), Bike2['road']
(<Bike.mtn: 2>, <Bike.road: 1>)
```

Assignment

enum_test.py

Syntax

Exception chaining (PEP 3134)

Chaining

Introduces:

- `__context__`
- `__cause__`
- `__traceback__`

Motivation

During the handling of one exception (exception A), it is possible that another exception (exception B) may occur... if this happens, exception B is propagated outward and exception A is lost. In order to debug the problem, it is useful to know about both exceptions. The `__context__` attribute retains this information

PEP 3134

__context__

```
>>> try:
...     answer = 1/0
... except ZeroDivisionError as e:
...     print(f"Exception: {str(e.__class__)}")
...     print(f"context: {e.__context__}")
...     print(f"cause: {e.__cause__}")
...     print(f"tb: {e.__traceback__}")
Exception: <class 'ZeroDivisionError'>
context: None
cause: None
tb: <traceback object at 0x111327588>
```

__context__

```
>>> def log(msg):
...     # pretend cloud service is down
...     raise SystemError("Logging not up")

>>> def divide_work(x, y):
...     try:
...         return x/y
...     except ZeroDivisionError as ex:
...         log("System is down")
```

__context__

```
>>> divide_work(5, 0)
Traceback (most recent call last):
  File "begpy.py", line 3, in divide_work
    return x/y
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "begpy.py", line 1, in <module>
    divide_work(5, 0)
  File "begpy.py", line 5, in divide_work
    log("System is down")
  File "begpy.py", line 2, in log
    raise SystemError("Logging not up")
SystemError: Logging not up
```

__context__

```
>>> try:
...     divide_work(6, 0)
... except Exception as e:
...     print(f"Exception: {e}")
...     print(f"context: {e.__context__}")
...     print(f"cause: {e.__cause__}")
...     print(f"tb: {e.__traceback__}")
Exception: Logging not up
context: division by zero
cause: None
tb: <traceback object at 0x111326488>
```

Motivation

Sometimes it can be useful for an exception handler to intentionally re-raise an exception, either to provide extra information or to translate an exception to another type. The `__cause__` attribute provides an explicit way to record the direct cause of an exception

PEP 3134

__cause__

Use `raise Exception from e` to chain:

```
>>> def log(msg):
...     print(msg)

>>> def divide_work(x, y):
...     try:
...         return x/y
...     except ZeroDivisionError as ex:
...         log("System is down")
...         raise ArithmeticError('bad math') from ex
```

__cause__

Use `raise Exception from e` to chain:

```
>>> divide_work(7, 0)
Traceback (most recent call last):
  File ..., line 3, in divide_work
    return x/y
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File ..., line 1, in <module>
    divide_work(7, 0)
  File ..., line 6, in divide_work
    raise ArithmeticError('bad math') from ex
ArithmeticError: bad math
```

__cause__

```
>>> try:
...     divide_work(8, 0)
... except Exception as e:
...     print(f"Exception: {e}")
...     print(f"context: {e.__context__}")
...     print(f"cause: {e.__cause__}")
System is down
Exception: bad math
context: division by zero
cause: division by zero
```

Motivation

Adding the `__traceback__` attribute to exception values makes all the exception information accessible from a single place.

PEP 3134

__traceback__

```
>>> try:
...     divide_work(9, 0)
... except Exception as e:
...     print(f"Exception: {str(e.__class__)}")
...     print(f"context: {e.__context__}")
...     print(f"cause: {e.__cause__}")
...     print(f"tb: {e.__traceback__}")
System is down
Exception: <class 'ArithmeticError'>
context: division by zero
cause: division by zero
tb: <traceback object at 0x1074aea08>
```

Exception Cleanup

Since tracebacks contain variable state, they are cleaned up following exception:

```
>>> try:
...     divide_work(9, 0)
... except Exception as e:
...     pass
>>> print('Exception', e)  # py2 works
Traceback (most recent call last):
   ...
NameError: name 'e' is not defined
```

Exception Handling

- Module specific exceptions can make discovery easier
- Be specific about what exceptions you handle
- Only handle what you can recover from

Assignment

`exception_test.py`

Extended Iterable Unpacking (PEP 3132 - 3.0)

Motivation

From PEP:

```
>>> a, *b, c = range(5)
>>> a
0
>>> c
4
>>> b
[1, 2, 3]
```

Notes

- Catch-all (*starred expression*) is a list not a tuple
- Can only have one starred expression (if not nested)
- Deals with left side of assignment

Unpacking Review

```
>>> a = 2
>>> b = 4
>>> a, b = b, a

>>> names = ['fred', 'george', 'luna', 'harry']
>>> first, rest = names[0], names[1:]
>>> first, rest
('fred', ['george', 'luna', 'harry'])

>>> person = ('Fred', 20, 'England',
...       ('Arthur', 'Molly'))
>>> name, age, loc, (dad, mom) = person
>>> name, dad
('Fred', 'Arthur')
```

Unpacking Review

```
>>> name_ages = [('Fred', 20), ('George', 20)]
>>> for i, (name, age) in enumerate(name_ages, 1):
...     print(f'{i}. {name:10} {age}')
1. Fred       20
2. George     20
```

Unpacking Review

```
>>> names = ['fred', 'george', 'luna', 'harry']
>>> first, *rest = names
>>> first, rest
('fred', ['george', 'luna', 'harry'])

>>> person = ('Fred', 20, 'England',
...         ('Arthur', 'Molly'))
>>> *ignore, (dad, mom) = person
>>> *ignore, ((dfirst, *d), (mfirst, *m)) = person
>>> dfirst
'A'
```

Gotcha

May need a trailing comma

```
>>> names = ['fred', 'george', 'luna', 'harry']
>>> *people = names
Traceback (most recent call last):
  ...
SyntaxError: starred assignment target must be
in a list or tuple
```

Gotcha

May need a trailing comma

```
>>> names = ['fred', 'george', 'luna', 'harry']
>>> *people, = names
>>> people
['fred', 'george', 'luna', 'harry']
```

Assignment

`unpack_test.py`

Additional Unpacking Generalizations (PEP 448 - 3.5)

Motivation

[Extend] usages of the * iterable unpacking operator and ** dictionary unpacking operators to allow unpacking in more positions, an arbitrary number of times

PEP 448

Combining Dictionaries

Old way:

```
>>> thing_colors = {'apple': 'red',
...     'pumpkin': 'orange'}
>>> more_colors = {}
>>> more_colors.update(thing_colors)
>>> more_colors['bike'] = 'blue'
>>> more_colors['apple'] = 'green'
>>> more_colors
{'apple': 'green', 'pumpkin': 'orange', 'bike': 'blue'}
```

Combining Dictionaries

New way (if repeated, last value wins):

```
>>> thing_colors = {'apple': 'red',
...     'pumpkin': 'orange'}
>>> more_colors = {**thing_colors,
...     'bike': 'blue', 'apple': 'green'}
>>> more_colors
{'apple': 'green', 'pumpkin': 'orange', 'bike':
'blue'}
```

Combining Dictionaries

Unpack can be in any location in the dictionary (here `apple` from `thing_colors` overrides green value):

```
>>> {'bike': 'blue', 'apple': 'green',
...         **thing_colors}
{'bike': 'blue', 'apple': 'red',
'pumpkin': 'orange'}
```

Multiple **'s

```
>>> def print_args(**kwargs):
...     for k, v in kwargs.items():
...         print(f"Key: {k:8} Val: {v}")

>>> thing_colors = {'apple': 'red',
...     'pumpkin': 'orange'}
>>> more_colors = {'bike': 'blue'}
>>> print_args(**thing_colors, hair='red', **more_colors)
Key: apple    Val: red
Key: pumpkin  Val: orange
Key: hair     Val: red
Key: bike     Val: blue
```

Multiple **'s

Can't repeat names in call:

```
>>> thing_colors = {'apple': 'red',
...     'pumpkin': 'orange'}
>>> more_colors = {'bike': 'blue', 'apple': 'green'}
>>> print_args(**thing_colors, hair='red', **more_colors)
Traceback (most recent call last):
  ...
TypeError: print_args() got multiple values for keyword
argument 'apple'
```

Creating Tuples

```
>>> name = 'matt'
>>> *name,
('m', 'a', 't', 't')
```

Creating Tuples

Comma may be required:

```
>>> name = 'matt'
>>> *name
Traceback (most recent call last):
    ...
SyntaxError: can't use starred expression
here
```

Tuples, Lists, and Sets

```
>>> name = 'matt'
>>> last = 'harrison'
>>> *name, *last
('m', 'a', 't', 't', 'h', 'a', 'r', 'r', 'i', 's', 'o',
'n')
>>> [*name, *last]
['m', 'a', 't', 't', 'h', 'a', 'r', 'r', 'i', 's', 'o',
'n']
>>> {*name, *last}
{'o', 'm', 'a', 's', 'i', 'r', 't', 'n', 'h'}
```

Function Arguments

```
>>> def summer(*args, **kwargs):
...     res = sum(args)
...     for v in kwargs.values():
...         res += v
...     return res
```

Function Arguments

```
>>> summer(1, 2, 3)
6
>>> summer(*[1, 2], 3)
6
>>> summer(*[1], 2, *[3])
6
>>> summer(*[1], 2, **{'v': 3}, y=0, **{'x': 0})
6
```

Vs Extended Unpacking

```
>>> name = 'matt'
>>> *letters, = name  # extended unpacking
>>> letters
['m', 'a', 't', 't']

>>> letters2 = *name,
>>> letters2
('m', 'a', 't', 't')
```

Assignment

`gen_unpack_test.py`

Other Changes

Iterators (Laziness)

- `map`, `filter`, `zip`
- `range` (iterable)
- Dict keys, items, values (views - reflect updates to the dict)

Strict Ordering

```
>>> 3 < '3'
Traceback (most recent call last):
  ...
TypeError: '<' not supported between
instances of 'int' and 'str'
```

Dictionary Order

In Python 3.6 dictionary insertion order is maintained (implementation detail in cPython). In 3.7, this is part of the language:

```
>>> d = {'name': 'matt'}
>>> d['age'] = 10
>>> d['address'] = '123 E Street'
>>> d
{'name': 'matt', 'age': 10, 'address': '123 E
Street'}
```

No Comprehension Name Leakage

```
>>> x = 10
>>> [x**2 for x in range(5)]
[0, 1, 4, 9, 16]
>>> x  # 4 in Py 2
10
```

Assignment

other_test.py

Conclusion

Python 3

Python 3 has awesome features

Thanks

`@__mharrison__` would love your feedback

Credits

- retro computer by Tinashe Mugayi - Noun Project
- python by Danil Polshin - Noun Project
- painting by Maxim Basinski - Noun Project
- Save Environment by Shastry from the Noun Project
- format by Aneeque Ahmed from the Noun Project
- emoji by Maxim Kulikov from the Noun Project
- Low Numbers by AlfredoCreates.com/icons & Flaticondesign.com from the Noun Project
- hierarchy by Creative Stall from the Noun Project
- hash function by dDara from the Noun Project
- Library by lastspark from the Noun Project
- Library by lastspark from the Noun Project
- Stopwatch by Creative Stall from the Noun Project
- Annotate by Garrett Knoll from the Noun Project
- omega by Jasmine Jones from the Noun Project
- tools by Dinosoft Labs from the Noun Project
- Suitcase by Federico Panzano from the Noun Project
- unpack box by arloenl evinniev from the Noun Project