A Python Quiz

Tibs

Produced for the January 2019 meeting of CamPUG.

Some of the conundrums are taken from the rather wonderful What the f*ck Python! by Satwik Kansal.

Note that Python 3 is assumed throughout.

1: Beginning with P [1 for each]

How many programming languages can you name that start with P, not including Python itself?

https://en.wikipedia.org/wiki/List_of_programming_languages lists

P, P", P4, PARI/GP, PCASTL, PCF, PDL, PEARL, PHP, PIKT, PILOT, PL-11, PL/0, PL/B, PL/C, PL/I, PL/M, PL/P, PL/SQL, PL360, PLANC, PLEX, PLEXIL, POP-11, POP-2, PPL, PROIV, PROMAL, PROSE modeling language, PROTEL, ParaSail, Pascal, PeopleCode, Perl, Perl 6, Pharo, Pico, Picolisp, Pict, Pig (programming tool), Pike, Pipelines, Pizza, Plankalkül, Planner, Plus, PortabIE, PostScript, PowerBuilder, PowerShell, Powerhouse, Pro*C, Processing, Processing.js, Prograph, Prolog, Promela, ProvideX, Pure, Pure Data, PureBasic, and Python itself

2: Quit [1]

What does this print?

```
>>> quit
```

```
>>> quit
Use quit() or Ctrl-D (i.e. EOF) to exit
```

3: Getting out [1 for each]

So how do you exit the Python prompt?

```
>>> quit()

>>> exit()
```

On Windows, the end-of-file character:

```
>>> <CTRL-Z>
```

On Unix, the endo-of-transmission character:

```
>>> <CTRL-D>
```

Or even:

```
>>> import sys; sys.exit()
```

4: To the... [1 for each]

What does the following print?

```
print(2**3, 2^3)
```

```
>>> print(2**3, 2^3)
8 1
```

The first is `power`, and the second `bitwise exclusive or`.

2 cubed is 8

Binary `10` exclusive or'ed with binary `11` is binary `01`.

5: Empty function [1]

What does this function return?

```
def fn():
    pass
```

It returns None.

```
>>> def fn():
...     pass
...
>>> fn()
>>> print(fn())
None
```

6: Empty function 2 [1]

What does this code do?

```
def fn():
    print('Aha')
fn
```

Well, nothing.

```
>>> def fn():
...     print('Aha')
...
>>> fn
<function fn at 0x10fbd7048>
```

7: Finally return [1]

What does this function return?

```
def fun():
    try:
        return 1
    finally:
        return 2
```

```
>>> def fun():
...     try:
...         return 1
...     finally:
...         return 2
...
>>> fun()
2
```

8: try/else/finally [1]

What does this function print?

```
try:
    print('try')
except Exception:
    print('except')
else:
    print('else')
finally:
    print('finally')
```

```
>>> try:
...     print('try')
... except Exception:
...     print('except')
... else:
...     print('else')
... finally:
...     print('finally')
...
try
else
finally
```

9: try/else/finally 2 [1]

So what does this function print?

```
try:
    print(f'try {1/0}')
except Exception:
    print('except')
else:
    print('else')
finally:
    print('finally')
```

```
>>> def fn():
...     try:
...         print(f'try {1/0}')
...     except Exception:
...         print('except')
...     else:
...         print('else')
...     finally:
...         print('finally')
...
>>> fn()
except
finally
```

10: Whose variable now? [1 per call of print]

What values should I expect to see printed out when I do the following?

```
class A:
    pass

A.x = 1
a = A()
print(A.x, a.x)
A.x = 2
print(A.x, a.x)
a.x = 3
print(A.x, a.x)
```

```
>>> class A:
...     pass
...
>>> A.x = 1
>>> a = A()
>>> print(A.x, a.x)
1 1
>>> A.x = 2
>>> print(A.x, a.x)
2 2
>>> a.x = 3
>>> print(A.x, a.x)
2 3
```

11: Format strings [1]

Which way of "quoting" is more useful, the first or second, and why?

```
print(f"The value is '{value}' vs {value!r}")
```

For a simple string `value` it may not be obvious:

```
>>> value = 'nine'
>>> print(f"The value is '{value}' vs {value!r}")
The value is 'nine' vs 'nine'
```

But if `value` is not a string the second makes this obvious:

```
>>> value = 1
>>> print(f"The value is '{value}' vs {value!r}")
The value is '1' vs 1
```

and it's also better if `value` is a string containing single quotes:

```
>>> value = "they're ready"
>>> print(f"The value is '{value}' vs {value!r}")
The value is 'they're ready' vs "they're ready"
```

...and if you're using old-fashioned %s formatting, then the equivalent is:

```
print(f"The value is '%s' vs %r" % (value, value))
```

12: Empty tuples [1]

How do you create an empty tuple?

```
>>> a = ()
>>> a
()
>>> type(a)
<class 'tuple'>
```

13: 1-Tuples [1]

So how do you create a tuple of one item?

```
>>> a = 1,
>>> a
(1,)
>>> type(a)
<class 'tuple'>
```

or:

```
>>> a = (1,)
>>> a
(1,)
>>> type(a)
<class 'tuple'>
```

But the following doesn't work:

```
>>> a = (1)
>>> a
1
>>> type(a)
<class 'int'>
```

14: Just what you expect [1]

What do the values get set to in:

```
tup = (1, 2, 3, 4)
a, *b, c = tup
d, *e = tup
```

```
>>> tup = (1, 2, 3, 4)

>>> a, *b, c = tup
>>> print(a, b, c)
1 [2, 3] 4

>>> d, *e = tup
>>> print(d, e)
1 [2, 3, 4]
```

15: Take care with % [1]

What does the following do?

```
>>> a = 1, 2
>>> print('a is %s' % a)
```

```
>>> a = 1, 2
>>> print('a is %s' % a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not all arguments converted during string formatting
```

Which is why you see people doing:

```
>>> print('a is %s' % (a,))
a is (1, 2)
```

or using:

```
>>> print(f'a is {a}')
a is (1, 2)
```

16: Logging [1]

Given:

```
import logging
logger = logging.getLogger(__name__)
a = 3
b = 4
```

Which is correct, the first, second or third, and why?

```
logger.info(f'A is {a} and B is {b}')
```

```
logger.info('A is %s and B is %r' % (a, b))
```

```
logger.info('A is %s and B is %r', a, b)
```

The third is correct:

```
logger.info('A is %s and B is %r', a, b)
```

as the logging callable will only construct the final string if the log message is actually output. In the other two examples, the final string is created when the `logger.info` call is made, even if the callable decides not to output anything.

17: More equal than expected [1]

After doing:

```
a = {}
a[5] = 'five'
a[5.0] = 'five point nought'
a[5.1] = 'five point one'
```

what does the dictionary contain?

```
>>> a = {}
>>> a[5] = 'five'
>>> a[5.0] = 'five point nought'
>>> a[5.1] = 'five point one'
>>> a
{5: 'five point nought', 5.1: 'five point one'}
```

Python regards 5 and 5.0 as equal (although not the same!)

```
>>> a[5.0]
'five point nought'
>>> 5 == 5.0
True
>>> 5 is 5.0
>>> 5 is 5.0
False
```

18: It's a what? [1]

OK, what does the dictionary contain after this?

```
b = {}
b[0] = 'nought'
b[1] = 'one'
b[2] = 'two'
b[False] = 'false'
b[True] = 'true'
```

```
>>> b = {}
>>> b[0] = 'nought'
>>> b[1] = 'one'
>>> b[2] = 'two'
>>> b[False] = 'false'
>>> b[True] = 'true'
>>> b
{0: 'false', 1: 'true', 2: 'two'}
```

For historical reasons, booleans are subtypes of integers.

```
>>> type(True)
<class 'bool'>
>>> isinstance(True, int)
True
>>> 1 == True
True
>>> True + True
2
```

19: Don't do this at home [2]

What does the following code print out?

```
def some_func(default_arg=[]):
    default_arg.append("ick")
    print(default_arg)

some_func()
some_func()
some_func(['aha'])
some_func()
```

```
>>> def some_func(arg=[]):
...     arg.append("ick")
...     print(arg)
...
>>> some_func()
['ick']
>>> some_func()
['ick', 'ick']
>>> some_func(['aha'])
['aha', 'ick']
>>> some_func()
['ick', 'ick', 'ick']
```

Perhaps we meant to do something more like:

```
def some_func(arg=None):
    if not arg:
        arg = []
    arg.append("ick")
    print(arg)
```

20: Mutation [1]

What values do you expect to remain in `list1` after doing:

```
list1 = [1, 2, 3, 4]
for item in list1:
  list1.remove(item)
```

```
>>> list1 = [1, 2, 3, 4]
>>> for item in list1:
...     list1.remove(item)
...
>>> print(list1)
[2, 4]
```

We look at the list, which contains `[1, 2, 3, 4]`, take its first value as `item`, and remove that, leaving us with `[2, 3, 4]`.

Then we look at the list, which now contains `[2, 3, 4]` and take its *second* value as `item`, and remove that, leaving us with `[2, 4]`.

There isn't a third value in `[2, 4]`, so we're done.

21: Enumeration [2]

After doing:

```
some_string = "wtf"
some_dict = {}
for i, some_dict[i] in enumerate(some_string):
    pass
```

what does `some_dict` contain?

```
>>> some_string = "wtf"
>>> some_dict = {}
>>> for i, some_dict[i] in enumerate(some_string):
...     pass
...
>>> print(some_dict)
{0: 'w', 1: 't', 2: 'f'}
```

It's as if we did:

```
i, some_dict[i] = 0, 'w'
i, some_dict[i] = 1, 't'
i, some_dict[i] = 2, 'f'
```

22: In or not in [1 for each]

What results do the following produce?

```
1 in [1,2,3]
[1,2] in [1,2,3]
'a' in 'abc'
'ab' in 'abc'
'' in 'abc'  # that's an empty string
```

```
>>> 1 in [1,2,3]
True
>>> [1,2] in [1,2,3]
False
>>> 'a' in 'abc'
True
>>> 'ab' in 'abc'
True
>>> '' in 'abc'
True
```

23: C does the same [1]

What does this print, and why?

```
print("Aha!""")
```

```
>>> print("Aha!""")
Aha!
```

is the same as:

```
>>> print("Aha!" "")
Aha!
```

which is the same as:

```
>>> print("Aha!" + "")
Aha!
```

24: Where did it go [2]

What happens when the following tries to print e?

```
e = 7
try:
    raise Exception()
except Exception as e:
    pass
print(e)
```

```
>>> e = 7
>>> try:
...   raise Exception()
... except Exception as e:
...   pass
...
>>> print(e)
NameError: name 'e' is not defined
```

When an except clause assigns an exception to a target (as here), that value is cleared at the end of the except clause. So the code acts like:

```
e = 7
try:
    raise Exception()
except Exception as e:
    try:
        pass
    finally:
        del e
print(e)
```

25: Follow through all the way [3]

After the following, what is a set to, and why?

```
a, b = a[b] = {}, 5
```

```
>>> a, b = a[b] = {}, 5
>>> print(a)
{5: ({...}, 5)}
```

Python defines assignment statements as:

```
(target_list "=")+ (expression_list | yield_expression)
```

and says:

> *An assignment statement evaluates the expression list (remember that this can be a single expression or a comma-separated list, the latter yielding a tuple) and assigns the single resulting object to each of the target lists, from left to right.*

So our example is the same as doing:

```
>>> exp = {}, 5
>>> print(exp)
{} 5

>>> a, b = exp
>>> print(a, b)
{} 5
```

Now, a refers to the same dictionary as in exp[0].

```
>>> a[5] = exp
>>> print(a)
{5: ({...}, 5)}
```

and we've got a recursive datastructure - the ... above indicates this.

```
>>> a is exp[0] is a[5][0] is a[5][0][5][0]  # and so on
True
```

26: Unicode [1]

What does `int('١٢٣٤٥٦٧٨٩')` return?

`int('١٢٣٤٥٦٧٨٩')` returns 123456789

In Python, Decimal characters include digit characters, and all characters that can be used to form decimal-radix numbers, e.g. `U+0660, ARABIC-INDIC DIGIT ZERO`.

27: Why do we need self? [2]

```
class A:
    def __init__(self, arg):
        self.arg = arg
    def incr(self):
        self.arg += 1
```

1. We need it as a method argument because it doesn't have to be called "self" - i.e., the programmer has to say what name to use.

   Note

   Also, if we want to be able to pass it in (so we can call a method as `<class_name>.<method_name>(<instance>, ...)`) then it helps to have an explicit place in the argument list for it. Although this is an edge case, and one could argue that it doesn't of itself *require* having self explicitly mentioned in the arguments.

2. We need it in a method body to differentiate between:

```
A.arg = 3
self.arg = 3
arg = 3
```