Solutions

Marking: **M1** a correct method, **A1** the correct answer, **B1** an independent mark.

Each answer shows the steps, then the **result**.

2.1

$\pi \times 1^2 + 2^2$

$= \mathbf{7.14}$

2.2

`TypeError`: an abstract class cannot be made

`TypeError` too: `Bad` is still abstract because `send` is missing

2.3

every function with this `if` chain must be found and edited for `"push"`

instead, each kind is a class with a `send` method

adding push is then one new `PushNotifier` class, and callers just call `n.send(...)`

2.4

$2 + 3 + 1 = \mathbf{6}$

`TypeError`: an `int` has no `len()`

3.1

(a)

```
class Notifier(ABC):
    @abstractmethod
    def send(self, user, msg): ...

class EmailNotifier(Notifier):
    def send(self, user, msg):
        smtp_send(user.email, msg)
        return True

class FakeNotifier(Notifier):
    def __init__(self):
        self.sent = []
    def send(self, user, msg):
        self.sent.append((user, msg))
        return True
```

the abstract method; the e-mail version; the fake records messages; both return `bool` as promised

(b) no real e-mails are sent and no network is needed; the test can check exactly which messages were sent

(c) `FakeNotifier.send`

3.2

(a) no: it demands more of its callers, a stronger precondition

(b) e.g. code that closes an account by withdrawing the whole balance fails for a savings account holding £800

(c) e.g. state the limit in the base interface (a `max_withdrawal()` method), or do not make savings a subclass

3.3

(a) if chains: all 6 functions; classes: 1 new class with its 6 methods, and no other code

(b) e.g. `class Payment(ABC):` with `@abstractmethod def charge(self, amount)` and `@abstractmethod def refund(self, amount)`

(c) adding an operation to the class design means editing all 4 classes; with functions it is one new function; so when operations change more often than types, the function design can be better: pick the design that keeps the common change in one place

exercises.lol exercises.com.cn